Verification of Relational Data-Centric Dynamic Systems with External Services

Babak Bagheri Hariri
Diego Calvanese
Marco Montali
Free Univ. of Bozen/Bolzano
lastname@inf.unibz.it

Giuseppe De Giacomo
Sapienza Università di Roma
degiacomo@dis.uniroma1.it

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

ABSTRACT

Data-centric dynamic systems are systems where both the process controlling the dynamics and the manipulation of data are equally central. Recently such kinds of systems are increasingly attracting the interest of the scientific community, especially in their variant called artifact-centric business processes. In this paper we study verification of (first-order) μ-calculus variants over *relational data-centric dynamic systems*, where data are represented by a full-fledged relational database, and the process is described in terms of atomic actions that evolve the database. The execution of such actions may involve calls to external services, providing fresh data inserted into the system. As a result such systems are typically infinite-state. We show that verification is undecidable in general, and we isolate notable cases, where decidability is achieved. Specifically we start by considering service calls that return values deterministically (depending only on passed parameters). We show that in a μ-calculus variant that preserves knowledge of objects appeared along a run we get decidability under the assumption that the fresh data introduced along a run are bounded, though they might not be bounded in the overall system. In fact we tie such a result to a notion related to weak acyclicity studied in data exchange. Then, we move to nondeterministic services where the assumption of data bounded run would result in a bound on the service calls that can be invoked during the execution and hence would be too restrictive. So we investigate decidability under the assumption that knowledge of objects is preserved only if they are continuously present. We show that if infinitely many values occur in a run but do not accumulate in the same state, then we get again decidability. We give syntactic conditions to avoid this accumulation through the novel notion of "generate-recall acyclicity", which takes into consideration that every service call activation generates new values that cannot be accumulated indefinitely.

1. INTRODUCTION

Data-centric dynamic systems (DCDSs) are systems where both the process controlling the dynamics and the manipulated data are equally central. Recently such kinds of systems are increasingly attracting the interest of the scientific community. In particular, the so called artifact-centric approach to modeling business processes has emerged, with the fundamental characteristic of considering both data and processes as first-class citizens in service design and analysis [32, 26, 18, 15, 36, 1]. This holistic view of data and processes together promises to avoid the notorious discrepancy between data modeling and process modeling of more traditional approaches that consider these two aspects separately [7, 6].

DCDSs are constituted by *(i)* a *data layer*, which is used to hold the relevant information to be manipulated by the system, and *(ii)* a *process layer* formed by the invokable *(atomic) actions* and a process based on them. Such a process characterizes the dynamic behavior of the system. Executing an action has effects on the data manipulated by the system, on the process state, and on the information exchanged with the external world.

DCDSs deeply challenge formal verification by requiring simultaneous attention to both data and processes: indeed, on the one hand they deal with full-fledged processes and require analysis in terms of sophisticated temporal properties [17]; on the other hand, the presence of possibly unbounded data makes the usual analysis based on model checking of finite-state systems impossible in general, since, when data evolution is taken into account, the whole system becomes infinite-state.

In this paper we study *relational* DCDSs, where data are represented by a full-fledged relational database, and the process is described in terms of atomic actions that evolve the database. The execution of such actions may involve calls to external services, providing fresh data inserted into the system. As a result such systems are infinite-state in general. In particular, actions are characterized using conditional effects. Effects are specified using first-order (FO) queries to extract from the current database the objects we want to persist in the next state, and using conjunctive queries on these objects to generate the facts that are true in the next state. In addition, to finalize the next state we call external services (function calls) that provide new information and objects coming from the external world.

On top of such a framework, we introduce powerful verification logics, which are FO variants of μ-calculus [29, 33, 22, 13]. μ-calculus is well known to be more expressive than virtually all temporal logics used in verification, including CTL, LTL, CTL*, PDL, and many others. Our approach is remarkably robust: while it is common to use simpler logics like CTL and LTL towards verification decidability, our decidability results hold for significantly more expressive μ-calculus variants, and thus carry over to all these other logics. Our variants of μ-calculus are based on first-order queries over data in the states of the DCDS, and allow for first-order quantification across states (within and across runs), though in a controlled way. No limitations whatsoever are instead put on the fixpoint formulae, which are the key element of the μ-calculus.

In particular we consider two variants of μ-calculus. The first variant is called $\mu\mathcal{L}_A$, and requires that first-order quantification across states be always bounded to the active domain of the state where the quantification is evaluated. This quantification mechanism indirectly preserves, at any point, knowledge of objects that appeared in the history so far, even if they disappeared in the meantime. The second variant, called $\mu\mathcal{L}_P$, restricts the first-order quantification in $\mu\mathcal{L}_A$ by requiring that only quantified object that are still present in the current domain are of interest as we move from one state to the next . That is, knowledge of objects is preserved only if they are continuously present. For these two logics we define novel notions of bisimulation, which we exploit to prove our results.

We show that verification of both $\mu\mathcal{L}_A$ and $\mu\mathcal{L}_P$ is undecidable in general. In fact we get undecidability even ruling out first-order quantification and branching time. However we isolate two notable decidable cases. Specifically we start by considering service calls that return values deterministically (depending only on passed parameters). We show that verification of $\mu\mathcal{L}_A$ properties is decidable under the assumption that the cardinality of fresh data introduced along each run is bounded (*run-bounded* DCDSs), though it need not be bounded across runs. Decidability is therefore not obvious, given that the logic permits quantification over values occurring across (potentially infinitely many) branching run continuations. Run-boundedness is a semantic property which we show undecidable to check, but for which we propose a sufficient syntactic condition related to the notion of weak acyclicity studied in data exchange [23]. Then, we move to nondeterministic services where same-argument service calls possibly return different values at different moments in time. To exploit the results on run-bounded DCDSs in this case we would have to limit the number of service calls that can be invoked during the execution, which would be a too restrictive condition on the form of DCDSs. So we focus on the above $\mu\mathcal{L}_P$ fragment of $\mu\mathcal{L}_A$. We show that if infinitely many values occur in a run but do not accumulate in the same state (our system is then called *state-bounded*) then $\mu\mathcal{L}_P$ verification is decidable. This comes as a pleasant surprise, given that when compared to run-boundedness, state-boundedness permits an additional kind of data unboundedness (*within* the run, as opposed to only *across* runs). State-boundedness is a semantic property as well, and we show that checking it is undecidable. We then give a novel syntactic condition, "generate-recall acyclicity", which suffices to enforce that if a service generates new values by being called an unbounded number of times, then these values cannot be accumulated ("recalled") indefinitely.

The rest of the paper is organized as follows. Sec. 2 introduces (relational) DCDS's. Sec. 3 introduces verification of DCDS's and the two variants of μ-calculus that we consider. Sec. 4 focus the analysis of DCDS's under the assumption that external service calls behave deterministically. Sec. 5 consider the case in which external service calls behave nondeterministically. Sec. 6 discusses the various notions introduced Sec. 7 reports on related work. Finally, Sec. 8 concludes the paper. All proofs are given in the appendix, which also includes a full-fledged example of a DCDS.

2. DATA-CENTRIC DYNAMIC SYSTEMS

In this section, we introduce the notion of *(relational) data-centric dynamic system*, or simply DCDS. A DCDS is a pair $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ formed by two interacting layers: a *data layer* $\mathcal{D}$ and a *process layer* $\mathcal{P}$ over it. Intuitively, the data layer keeps all the data of interest, while the process layer modifies and evolves such data. We keep the structure of both layers to the minimum, in particular we do not distinguish between various possible components providing the data, nor those providing the subprocesses running concurrently. Indeed the framework can be further detailed in several directions, while keeping the results obtained here (cf. Section 6).

2.1 Data Layer

The data layer represents the information of interest in our application. It is constituted by a relational schema $\mathcal{R}$ equipped with equality constraints[1] $\mathcal{E}$, e.g., to state keys of relations, and an initial database instance $\mathcal{I}_0$, which conforms to the relational schema and the equality constraints. The values stored in this database belong to a predefined, possibly infinite, set $\mathcal{C}$ of constants. These constants are interpreted as themselves, blurring the distinction between constants and values. We will use the two terms interchangeably.

Given a database instance $\mathcal{I}$, its active domain $\textsc{adom}(\mathcal{I})$ is the subset of $\mathcal{C}$ such that $c \in \textsc{adom}(\mathcal{I})$ if and only if c occurs in $\mathcal{I}$.

A *data layer* is a tuple $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ where:

- $\mathcal{C}$ is a countably infinite set of constants/values.
- $\mathcal{R} = \{R_1, \ldots, R_n\}$ is a database schema, constituted by a finite set of relation schemas.
- $\mathcal{E}$ is a finite set $\{\mathcal{E}_1, \ldots, \mathcal{E}_m\}$ of equality constraints. Each $\mathcal{E}_i$ has the form

$$Q_i \rightarrow \textstyle\bigwedge_{j=1,\ldots,k} z_{ij} = y_{ij},$$

  where Q_i is a domain independent FO query over $\mathcal{R}$ using constants from the active domain $\textsc{adom}(\mathcal{I}_0)$ of $\mathcal{I}_0$ and whose free variables are $\vec{x}$, and z_{ij} and y_{ij} are either variables in $\vec{x}$ or constants in $\textsc{adom}(\mathcal{I}_0)$.[2]
- $\mathcal{I}_0$ is a database instance that represents the initial state of the data layer, which conforms to the schema $\mathcal{R}$ and *satisfies* the constraints $\mathcal{E}$: namely, for each constraint $Q_i \rightarrow \bigwedge_{j=1,\ldots,k} z_{ij} = y_{ij}$ and for each tuple (i.e., substitution for the free variables) $\theta \in ans(Q_i, \mathcal{I})$, it holds that $z_{ij}\theta = y_{ij}\theta$.[3]

2.2 Process Layer

The process layer constitutes the progression mechanism for the DCDS. We assume that at every time the current instance of the data layer can be arbitrarily queried, and can be updated through action executions, possibly involving external service calls to get new values form the environment. Hence the process layer is composed of three main notions: actions, which are the atomic progression steps for the data layer; external services, which can be called during the execution of actions; and processes, which are essentially nondeterministic programs that use actions as atomic instructions. While we require the execution of actions to be sequential, we do not impose any such constraints on processes, which in principle can be formed by several concurrent branches, including fork, join, and so on. Concurrency is to be interpreted by interleaving and hence reduced to nondeterminism, as often done in formal

[1]Other kinds of constraints can also be included without affecting the results reported here (cf. Section 6).

[2]For convenience, and without loss of generality, we assume that all constants used inside formulae appear in $\mathcal{I}_0$.

[3]We use the notation $t\theta$ (resp., $\varphi\theta$) to denote the term (resp., the formula) obtained by applying the substitution θ to t (resp., φ). Furthermore, given a FO query Q and a database instance $\mathcal{I}$, the *answer* $ans(Q, \mathcal{I})$ to Q over $\mathcal{I}$ is the set of assignments θ from the free variables of Q to the domain of $\mathcal{I}$, such that $\mathcal{I} \models Q\theta$. We treat $Q\theta$ as a boolean query, and with some abuse of notation, we say $ans(Q\theta, \mathcal{I}) \equiv \mathtt{true}$ if and only if $\mathcal{I} \models Q\theta$.

verification [4, 22]. There can be many ways to provide the control flow specification for processes. Here we adopt a simple rule-based mechanism, but our results can be immediately generalized to any process formalism whose processes control flow is finite-state. Notice that this does not imply that the transition system associated to a process over the data layer is finite-state as well, since the data manipulated in the data layer may grow over time in an unbounded way.

Formally, a process layer $\mathcal{P}$ over a data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$, is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$ where:

- $\mathcal{F}$ is a finite set of *functions*, each representing the interface to an *external service*. Such services can be called, and as a result the function is activated and the answer is produced. How the result is actually computed is *unknown* to the DCDS since the services are indeed external.
- $\mathcal{A}$ is a finite set of *actions*, whose execution progresses the data layer, and may involve external service calls.
- ϱ is a finite set of *condition-action rules* that form the specification of the overall *process*, which tells at any moment which actions can be executed.

An *action* $\alpha \in \mathcal{A}$ has the form

$$\alpha(p_1, \ldots, p_n) : \{e_1, \ldots, e_m\},$$

where: *(i)* $\alpha(p_1, \ldots, p_n)$ is the *signature* of the action, constituted by a name α and a sequence $p_1, \ldots, p_n$ of *input parameters* that need to be substituted with values for the execution of the action, and *(ii)* $\{e_1, \ldots, e_m\}$, also denoted as $\text{EFFECT}(\alpha)$, is a set of *effect specifications*, whose specified effects are assumed to take place simultaneously. Each e_i has the form $q_i^+ \wedge Q_i^- \rightsquigarrow E_i$, where:

- $q_i^+ \wedge Q_i^-$ is a query over $\mathcal{R}$ whose terms are variables $\vec{x}$, action parameters, and constants from $\text{ADOM}(\mathcal{I}_0)$. The query q_i^+ is a UCQ, and the query Q_i^- is an arbitrary FO formula whose free variables are included in those of q_i^+. Intuitively, q_i^+ selects the tuples to instantiate the effect, and Q_i^- filters away some of them.
- E_i is the effect, i.e., a set of facts for $\mathcal{R}$, which includes as terms: terms in $\text{ADOM}(\mathcal{I}_0)$, input parameters, free variables of q_i^+, and in addition Skolem terms formed by applying a function $f \in \mathcal{F}$ to one of the previous kinds of terms. Such Skolem terms involving functions represent external service calls and are interpreted so as to return a value chosen by an external user/environment when executing the action.

The *process* ϱ is a finite set of *condition-action rules*, of the form $Q \mapsto \alpha$, where α is an action in $\mathcal{A}$ and Q is a FO query over $\mathcal{R}$ whose free variables are exactly the parameters of α, and whose other terms can be either quantified variables or constants in $\text{ADOM}(\mathcal{I}_0)$.

For a detailed example of a DCDS we refer to Appendix E.

2.3 Semantics via Transition System

The semantics of a DCDS is defined in terms of a possibly infinite transition system whose states are labeled by databases. Such a transition system represents all possible computations that the process layer can do on the data layer. A transition system Υ is a tuple of the form $\langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$, where:

- Δ is a countably infinite set of values;
- $\mathcal{R}$ is a database schema;
- Σ is a set of states;
- $s_0 \in \Sigma$ is the initial state;
- db is a function that, given a state $s \in \Sigma$, returns the database associated to s, which is made up of values in Δ and conforms to $\mathcal{R}$;
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between pairs of states.

$$
\begin{aligned}
(Q)^{\Upsilon}_{v,V} &= \{s \in \Sigma \mid ans(Qv, db(s))\} \\
(\neg\Phi)^{\Upsilon}_{v,V} &= \Sigma - (\Phi)^{\Upsilon}_{v,V} \\
(\Phi_1 \wedge \Phi_2)^{\Upsilon}_{v,V} &= (\Phi_1)^{\Upsilon}_{v,V} \cap (\Phi_2)^{\Upsilon}_{v,V} \\
(\exists x.\Phi)^{\Upsilon}_{v,V} &= \{s \in \Sigma \mid \exists t.t \in \Delta \text{ and } s \in (\Phi)^{\Upsilon}_{v[x/t],V}\} \\
(\langle - \rangle \Phi)^{\Upsilon}_{v,V} &= \{s \in \Sigma \mid \exists s'.s \Rightarrow s' \text{ and } s' \in (\Phi)^{\Upsilon}_{v,V}\} \\
(Z)^{\Upsilon}_{v,V} &= V(Z) \\
(\mu Z.\Phi)^{\Upsilon}_{v,V} &= \bigcap \{\mathcal{S} \subseteq \Sigma \mid (\Phi)^{\Upsilon}_{v,V[Z/\mathcal{S}]} \subseteq \mathcal{S}\}
\end{aligned}
$$

Figure 1: Semantics of $\mu\mathcal{L}$.

In order to precisely build the transition system associated to a DCDS, we need to better characterize the behavior of the external services, which are called in the effects of actions. This is done in Sections 4 and 5.

3. VERIFICATION

To specify dynamic properties over a DCDS, we use μ-calculus [22, 35, 13], one of the most powerful temporal logics for which model checking has been investigated in the finite-state setting. Indeed, such a logic is able to express both linear time logics such as LTL and PSL, and branching time logics such as CTL and CTL* [17]. The main characteristic of μ-calculus is the ability of expressing directly least and greatest fixpoints of (predicate-transformer) operators formed using formulae relating the current state to the next one. By using such fixpoint constructs one can easily express sophisticated properties defined by induction or co-induction. This is the reason why virtually all logics used in verification can be considered as fragments of μ-calculus. From a technical viewpoint, μ-calculus separates local properties, i.e., properties asserted on the current state or on states that are immediate successors of the current one, and properties that talk about states that are arbitrarily far away from the current one [13]. The latter are expressed through the use of fixpoints.

In this work, we use a first-order variant of the μ-calculus [33], called $\mu\mathcal{L}$ and defined as follows:

$$\Phi ::= Q \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\Phi \mid \langle - \rangle \Phi \mid Z \mid \mu Z.\Phi$$

where Q is a possibly open FO query, and Z is a second order predicate variable (of arity 0). We make use of the following abbreviations: $\forall x.\Phi = \neg(\exists x.\neg\Phi)$, $\Phi_1 \vee \Phi_2 = \neg(\neg\Phi_1 \wedge \neg\Phi_2)$, $[-]\Phi = \neg\langle - \rangle\neg\Phi$, and $\nu Z.\Phi = \neg\mu Z.\neg\Phi[Z/\neg Z]$.

As usual in μ-calculus, formulae of the form $\mu Z.\Phi$ (and $\nu Z.\Phi$) must obey to the *syntactic monotonicity* of Φ wrt Z, which states that every occurrence of the variable Z in Φ must be within the scope of an even number of negation symbols. This ensures that the least fixpoint $\mu Z.\Phi$ (as well as the greatest fixpoint $\nu Z.\Phi$) always exists.

Since $\mu\mathcal{L}$ also contains formulae with both individual and predicate free variables, given a transition system Υ, we introduce an individual variable valuation v, i.e., a mapping from individual variables x to Δ, and a predicate variable valuation V, i.e., a mapping from predicate variables Z to subsets of Σ. With these three notions in place, we assign meaning to formulae by associating to Υ, v, and V an *extension function* $(\cdot)^{\Upsilon}_{v,V}$, which maps formulae to subsets of Σ. Formally, the extension function $(\cdot)^{\Upsilon}_{v,V}$ is defined inductively as shown in Figure 1.

EXAMPLE 3.1. An example of $\mu\mathcal{L}$ formula is:

$$\exists x_1, \ldots, x_n. \bigwedge_{i \neq j} x_i \neq x_j \wedge \bigwedge_{i \in \{1,\ldots,n\}} \mu Z.[Stud(x_i) \vee \langle - \rangle Z] \quad (1)$$

The formula asserts that there are at least n distinct objects/values, each of which eventually denotes a student along some execution path. Notice that the formula does not imply that all of these students will be in the same state, nor that they will all occur in a single run. It only says that in the entire transition systems there are (at least) n distinct students. ■

When Φ is a closed formula, $(\Phi)^{\Upsilon}_{v,V}$ depends neither on v nor on V, and we denote the extension of Φ simply by $(\Phi)^{\Upsilon}$. We say that a closed formula Φ holds in a state $s \in \Sigma$ if $s \in (\Phi)^{\Upsilon}$. In this case, we write $\Upsilon, s \models \Phi$. We say that a closed formula Φ holds in Υ, denoted by $\Upsilon \models \Phi$, if $\Upsilon, s_0 \models \Phi$, where s_0 is the initial state of Υ. We call *model checking* verifying whether $\Upsilon \models \Phi$ holds.

In particular we are interested in formally verifying properties of a DCDS. Given the transition system $\Upsilon_{\mathcal{S}}$ of a DCDS $\mathcal{S}$ and a dynamic property Φ expressed in $\mu\mathcal{L}$,[4] we say that $\mathcal{S}$ *verifies* Φ if

$$\Upsilon_{\mathcal{S}} \models \Phi.$$

The challenging point is that $\Upsilon_{\mathcal{S}}$ is in general-infinite state, so we would like to devise a finite-state transition system which is a faithful abstraction of $\Upsilon_{\mathcal{S}}$, in the sense that it preserves the truth value of all $\mu\mathcal{L}$ formulae. Unfortunately, this program is doomed right from the start if we insist on using full $\mu\mathcal{L}$ as the verification formalism. Indeed formulae of the form (1) defeat any kind of finite-state transition system. So next we introduce two interesting sublogics of $\mu\mathcal{L}$ that serve better our objective.

3.1 History Preserving Mu-Calculus

The first fragment of $\mu\mathcal{L}$ that we consider is $\mu\mathcal{L}_A$, which is characterized by the assumption that quantification over individuals is restricted to individuals that are present in the current database. To enforce such a restriction, we introduce a special predicate $\text{LIVE}(x)$, which states that x belongs to the current active domain. The logic $\mu\mathcal{L}_A$ is defined as follows:

$$\Phi ::= Q \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\text{LIVE}(x) \wedge \Phi \mid \langle - \rangle \Phi \mid Z \mid \mu Z.\Phi$$

We make use of the usual abbreviation, including $\forall x.\text{LIVE}(x) \rightarrow \Phi = \neg(\exists x.\text{LIVE}(x) \wedge \neg\Phi)$. Formally, the extension function $(\cdot)^{\Upsilon}_{v,V}$ is defined inductively as in Figure 1, with the new special predicate $\text{LIVE}(x)$ interpreted as follows:

$$(\text{LIVE}(x))^{\Upsilon}_{v,V} = \{s \in \Sigma \mid x/d \in v \text{ implies } d \in \text{ADOM}(db(s))\}$$

EXAMPLE 3.2. As an example, consider the following $\mu\mathcal{L}_A$ formula:

$$\nu X.(\forall x.\text{LIVE}(x) \wedge Stud(x) \rightarrow \mu Y.(\exists y.\text{LIVE}(y) \wedge Grad(x,y) \vee \langle - \rangle Y) \wedge [-]X),$$

which states that, along every path, it is always true, for each student x, that there exists an evolution that eventually leads to a graduation of the student (with some final mark y). ■

We are going to show that under suitable conditions we can get a faithful finite abstraction for a DCDS that preserves all formulae of $\mu\mathcal{L}_A$, and hence enables us in principle to use standard model checking techniques. Towards this goal, we introduce a notion of bisimulation that is suitable for the kind of transition systems we consider here. In particular, we have to take into account that the two transition systems are over different data domains, and hence we have to consider the correspondence between the data in the two transition systems and how such data evolve over time. To do so, we introduce the following notions.

Given two domains Δ_1 and Δ_2, a *partial bijection* h between Δ_1 and Δ_2 is a bijection between a subset of Δ_1 and Δ_2. Given a partial function $f : S \rightarrow S'$, we denote with $\text{DOM}(f)$ the domain of f, i.e., the set of elements in S on which f is defined, and with $\text{IM}(f)$ the image of f, i.e., the set of elements s' in S' such that $s' = f(s)$ for some $s \in S$. A partial bijection h' *extends* h if $\text{DOM}(h) \subseteq \text{DOM}(h')$ (or equivalently $\text{IM}(h) \subseteq \text{IM}(h')$) and $h'(x) = h(x)$ for all $x \in \text{DOM}(h)$ (or equivalently $h'^{-1}(y) = h^{-1}(y)$ for all $y \in \text{IM}(h)$). Let db_1 and db_2 be two databases over two domains Δ_1 and Δ_2 respectively, both conforming to the same schema $\mathcal{R}$. We say that a partial bijection h *induces an isomorphism* between db_1 and db_2 if $\text{ADOM}(db_1) \subseteq \text{DOM}(h)$, $\text{ADOM}(db_2) \subseteq \text{IM}(h)$, and h projected on $\text{ADOM}(db_1)$ is an isomorphism between db_1 and db_2.

Given two transition systems $\Upsilon_1 = \langle \Delta_1, \mathcal{R}, \Sigma_1, s_{01}, db_1, \Rightarrow_1 \rangle$ and $\Upsilon_2 = \langle \Delta_2, \mathcal{R}, \Sigma_2, s_{02}, db_2, \Rightarrow_2 \rangle$, and the set H of partial bijections between Δ_1 and Δ_2, a *history preserving bisimulation* between Υ_1 and Υ_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times H \times \Sigma_2$ such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$ implies that:

1. h is a partial bijection between Δ_1 and Δ_2 that induces an isomorphism between $db_1(s_1)$ and $db_2(s_2)$;
2. for each s_1', if $s_1 \Rightarrow_1 s_1'$ then there is an s_2' with $s_2 \Rightarrow_2 s_2'$ and a bijection h' that extends h, such that $\langle s_1', h', s_2' \rangle \in \mathcal{B}$.
3. for each s_2', if $s_2 \Rightarrow_2 s_2'$ then there is an s_1' with $s_1 \Rightarrow_1 s_1'$ and a bijection h' that extends h, such that $\langle s_1', h', s_2' \rangle \in \mathcal{B}$.

A state $s_1 \in \Sigma_1$ is *history preserving bisimilar* to $s_2 \in \Sigma_2$ *wrt a partial bijection* h, written $s_1 \approx_h s_2$, if there exists a history preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$. A state $s_1 \in \Sigma_1$ is *history preserving bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \approx s_2$, if there exists a partial bijection h and a history preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$. A transition system Υ_1 is *history preserving bisimilar* to Υ_2, written $\Upsilon_1 \approx \Upsilon_2$, if there exists a partial bijection h_0 and a history preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_{01}, h_0, s_{02} \rangle \in \mathcal{B}$. The next theorem gives us the classical invariance result of μ-calculus wrt bisimulation, in our setting.

THEOREM 3.1. *Consider two transition systems Υ_1 and Υ_2 such that $\Upsilon_1 \approx \Upsilon_2$. Then for every $\mu\mathcal{L}_A$ closed formula Φ , we have:*

$$\Upsilon_1 \models \Phi \text{ if and only if } \Upsilon_2 \models \Phi.$$

3.2 Persistence Preserving Mu-Calculus

The second fragment of $\mu\mathcal{L}$ that we consider is $\mu\mathcal{L}_P$, which further restricts $\mu\mathcal{L}_A$ by requiring that individuals over which we quantify must continuously persist along the system evolution for the quantification to take effect.

With a slight abuse of notation, in the following we write $\text{LIVE}(x_1, \ldots, x_n) = \bigwedge_{i \in \{1,\ldots,n\}} \text{LIVE}(x_i)$.

The logic $\mu\mathcal{L}_P$ is defined as follows:

$$\Phi ::= Q \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\text{LIVE}(x) \wedge \Phi \mid \langle - \rangle(\text{LIVE}(\vec{x}) \wedge \Phi) \mid [-](\text{LIVE}(\vec{x}) \wedge \Phi) \mid Z \mid \mu Z.\Phi$$

where Q is a possibly open FO query, Z is a second order predicate variable, and the following assumption holds: in $\langle - \rangle(\text{LIVE}(\vec{x}) \wedge \Phi)$

[4] We remind the reader that, without loss of generality, we assume that all constants used inside formulae Φ appear in the initial database instance of the DCDS.

and $[-](\text{LIVE}(\vec{x}) \wedge \Phi)$, the variables $\vec{x}$ are exactly the free variables of Φ, with the proviso that we substitute to each bounded predicate variable Z in Φ its bounding formula $\mu Z.\Phi'$. We use the usual abbreviations, including: $\langle - \rangle(\text{LIVE}(\vec{x}) \rightarrow \Phi) = \neg[-](\text{LIVE}(\vec{x}) \wedge \neg\Phi)$ and $[-](\text{LIVE}(\vec{x}) \rightarrow \Phi) = \neg\langle - \rangle(\text{LIVE}(\vec{x}) \wedge \neg\Phi)$. Intuitively, the use of $\text{LIVE}(\cdot)$ in $\mu\mathcal{L}_P$ ensures that individuals are only considered if they persist along the system evolution, while the evaluation of a formula with individuals that are not present in the current database trivially leads to false or true (depending on the use of negation).

EXAMPLE 3.3. Getting back to the example above, its variant in $\mu\mathcal{L}_P$ is

$$\nu X.(\forall x.\text{LIVE}(x) \wedge Stud(x) \rightarrow \mu Y.(\exists y.\text{LIVE}(y) \wedge Grad(x,y) \vee \langle - \rangle(\text{LIVE}(x) \wedge Y)) \wedge [-]X)$$

which states that, along every path, it is always true, for each student x, that there exists an evolution in which x persists in the database until she eventually graduates (with some final mark y). Formula

$$\nu X.(\forall x.\text{LIVE}(x) \wedge Stud(x) \rightarrow \mu Y.(\exists y.\text{LIVE}(y) \wedge Grad(x,y) \vee \langle - \rangle(\text{LIVE}(x) \rightarrow Y)) \wedge [-]X)$$

instead states that, along every path, it is always true, for each student x, that there exists an evolution in which either x is not persisted, or becomes eventually graduated (with final mark y). ■

The bisimulation relation that captures $\mu\mathcal{L}_P$ is as follows. Given two transition systems $\Upsilon_1 = \langle \Delta_1, \mathcal{R}, \Sigma_1, s_{01}, db_1, \Rightarrow_1 \rangle$ and $\Upsilon_2 = \langle \Delta_2, \mathcal{R}, \Sigma_2, s_{02}, db_2, \Rightarrow_2 \rangle$, and the set H of partial bijections between Δ_1 and Δ_2, a *persistence preserving bisimulation* between Υ_1 and Υ_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times H \times \Sigma_2$ such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$ implies that:

1. h is an isomorphism between $db_1(s_1)$ and $db_2(s_2)$;[5]
2. for each s_1', if $s_1 \Rightarrow_1 s_1'$ then there exists an s_2' with $s_2 \Rightarrow_2 s_2'$ and a bijection h' that extends $h|_{\text{ADOM}(db_1(s_1)) \cap \text{ADOM}(db_1(s_1'))}$, such that $\langle s_1', h', s_2' \rangle \in \mathcal{B}$;[6]
3. for each s_2', if $s_2 \Rightarrow_2 s_2'$ then there exists an s_1' with $s_1 \Rightarrow_1 s_1'$ and a bijection h' that extends $h|_{\text{ADOM}(db_1(s_1)) \cap \text{ADOM}(db_1(s_1'))}$, such that $\langle s_1', h', s_2' \rangle \in \mathcal{B}$.

We say that a state $s_1 \in \Sigma_1$ is *persistence preserving bisimilar* to $s_2 \in \Sigma_2$ *wrt a partial bijection* h, written $s_1 \sim_h s_2$, if there exists a persistence preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$. A state $s_1 \in \Sigma_1$ is *persistence preserving bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim s_2$, if there exists a partial bijection h and a persistence preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$. A transition system Υ_1 is *persistence preserving bisimilar* to Υ_2, written $\Upsilon_1 \sim \Upsilon_2$, if there exists a partial bijection h_0 and a persistence preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_{01}, h_0, s_{02} \rangle \in \mathcal{B}$. The next theorem shows the invariance of $\mu\mathcal{L}_P$ under this notion of bisimulation.

THEOREM 3.2. *Consider two transition systems Υ_1 and Υ_2 such that $\Upsilon_1 \sim \Upsilon_2$. Then for every $\mu\mathcal{L}_P$ closed formula Φ, we have:*

$$\Upsilon_1 \models \Phi \text{ if and only if } \Upsilon_2 \models \Phi.$$

[5]Notice that this implies $\text{DOM}(h) = \text{ADOM}(db_1(s_1))$ and $\text{IM}(h) = \text{ADOM}(db_2(s_2))$.

[6] Given a set D, we denote by $f|_D$ the *restriction* of f to D, i.e., $\text{DOM}(f|_D) = \text{DOM}(f) \cap D$, and $f|_D(x) = f(x)$ for every $x \in \text{DOM}(f) \cap D$.

4. DETERMINISTIC SERVICES

Now we turn back to the semantics of DCDSs, and analyze them under the assumption that external services behave deterministically. This means that the evaluation of functions $f \in \mathcal{F}$, representing the service interfaces in the process layer, is independent from the moment in which the function is called: whenever an external service is called twice with the same parameters, it must return the same value. So, for example, if the function invocation $f(a)$ returned b at a certain time, then in all successive moments the call $f(a)$ will return b again. In particular, *stateless* services can be modeled with deterministic service calls.

Under this characterization of the services we can now define the transition system of a DCDS. We call such a transition system "concrete" transition system to avoid confusion with an "abstract" transition system that we are going to introduce for our verification technique.

4.1 Semantics

Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$.

First we focus on what is needed to characterize the states of the concrete transition system. One such state obviously needs to maintain the current instance of the data layer. This instance is a database made up of constants in $\mathcal{C}$, which conforms to the schema $\mathcal{R}$ and satisfies the equality constraints in $\mathcal{E}$. Together with the current instance, however, we also need to remember all answers we had so far when calling the external services.

To meet the requirement that service calls behave deterministically, the states of the transition system keep track of all results of the service calls made so far, in the form of equalities between Skolem terms involving functions in $\mathcal{F}$ and having as arguments constants and returned values in $\mathcal{C}$.[7] More precisely, we define the set of (Skolem terms representing) service calls as $\mathbb{SC} = \{f(v_1, \ldots, v_n) \mid f/n \in \mathcal{F} \text{ and } \{v_1, \ldots, v_n\} \subseteq \mathcal{C}\}$, where f/n stands for a function f arity n. Then we introduce a *service call map*, which is a partial function $\mathcal{M} : \mathbb{SC} \rightarrow \mathcal{C}$.

Now we are ready to formally define states of the concrete transition system. A *concrete state*, or simply state, is a pair $\langle \mathcal{I}, \mathcal{M} \rangle$, where $\mathcal{I}$ is a relational instance of $\mathcal{R}$ over $\mathcal{C}$ satisfying each equality constraint in $\mathcal{E}$, and $\mathcal{M}$ is a service call map. The *initial concrete state* is $\langle \mathcal{I}_0, \emptyset \rangle$.

Next we look at the result of executing an action in a state. For this it is convenient to denote the database instance $\mathcal{M}(E)$ obtained by applying a service call map $\mathcal{M}$ to a set E of facts including only constants in $\mathcal{C}$ or terms in $\text{DOM}(\mathcal{M})$. Namely, we define $\mathcal{M}(E)$ as the application of $\mathcal{M}$ to all the terms appearing in E where constants are preserved. Formally, $\mathcal{M}(E) = \{R(c_1 \ldots, c_n) \mid R(t_1, \ldots, t_n) \in E \text{ and } c_i = t_i \text{ if } t_i \in \mathcal{C} \text{ and } \mathcal{M}(t_i) = c_i \text{ if } t_i \in \text{DOM}(\mathcal{M}) \text{ for } i \in \{1, \ldots, n\}\}$.

Let α be an action in $\mathcal{A}$ of the form $\alpha(p_1, \ldots, p_m) : \{e_1, \ldots, e_m\}$ with $e_i = q_i^+ \wedge Q_i^- \rightsquigarrow E_i$. The parameters for α are guarded by the condition-action rule $Q \mapsto \alpha$ in ϱ. Let σ be a substitution for the input parameters $p_1, \ldots, p_m$ with values taken from $\mathcal{C}$. We say that σ is *legal* for α in state $\langle \mathcal{I}, \mathcal{M} \rangle$ if $\langle p_1, \ldots, p_m \rangle \sigma \in ans(Q, \mathcal{I})$.

[7]Notice that, we have no knowledge of the specific functions adopted by the external services, and we simply assume that such functions return some value from $\mathcal{C}$. We are going to have different executions of the system corresponding to each way to assign values to the Skolem terms representing the service calls.

Concrete action execution. To capture what happens when α is executed in a state using a substitution σ for its parameters, we introduce a transition relation $\text{EXEC}_{\mathcal{S}}$ between states, called *concrete execution* of $\alpha\sigma$, such that $\langle\langle\mathcal{I},\mathcal{M}\rangle,\alpha\sigma,\langle\mathcal{I}',\mathcal{M}'\rangle\rangle \in \text{EXEC}_{\mathcal{S}}$ if the following holds:

1. σ is a legal parameter assignment for α in state $\langle\mathcal{I},\mathcal{M}\rangle$,
2. $\mathcal{M}' = \text{SERVICECALLS}(\mathcal{I},\alpha\sigma,\mathcal{M})$,
3. $\mathcal{I}' = \mathcal{M}'(\text{DO}(\mathcal{I},\alpha\sigma))$, and
4. $\mathcal{I}'$ satisfies $\mathcal{E}$,

where DO() and SERVICECALLS() are defined as follows.

$$\text{DO}(\mathcal{I},\alpha\sigma) = \bigcup_{q_i^+\wedge Q_i^- \rightsquigarrow E_i \in \text{EFFECT}(\alpha)} \ \bigcup_{\theta \in ans((q_i^+\wedge Q_i^-)\sigma,\mathcal{I})} E_i\sigma\theta$$

applies the action α to $\mathcal{I}$, using σ as the assignment for its parameters. The returned instance is the union of the results of applying the effects specifications $\text{EFFECT}(\alpha)$, where the result of each effect specification $q_i^+ \wedge Q_i^- \rightsquigarrow E_i$ is, in turn, the set of facts $E_i\sigma\theta$ obtained from $E_i\sigma$ grounded on all the assignments θ that satisfy the query $q_i^+ \wedge Q_i^-$ over $\mathcal{I}$.

$$\text{SERVICECALLS}(\mathcal{I},\alpha\sigma,\mathcal{M}) = \mathcal{M} \cup \{t \mapsto \text{PICKVALUE}(\mathcal{C}) \mid t \text{ occurring in } \text{DO}(\mathcal{I},\alpha\sigma) \text{ and not in } \text{DOM}(\mathcal{M})\}$$

nondeterministically generates all possible values that can be returned by the service calls, guaranteeing that external services behave in a deterministic manner. More specifically, all the service calls already contained in $\mathcal{M}$ are maintained, while new service calls are nondeterministically bound to an arbitrary value $\text{PICKVALUE}(\mathcal{C})$ taken from $\mathcal{C}$ (which will be the values assumed by such service calls in $\mathcal{M}$ from now on in the execution).

Concrete transition system. The *concrete transition system* $\Upsilon_{\mathcal{S}}$ for $\mathcal{S}$ is a possibly infinite-state transition system $\langle\mathcal{C},\mathcal{R},\Sigma,s_0,db,\Longrightarrow\rangle$ where $s_0 = \langle\mathcal{I}_0,\emptyset\rangle$ and db is such that $db(\langle\mathcal{I},\mathcal{M}\rangle) = \mathcal{I}$. Specifically, we define by simultaneous induction Σ and $\Longrightarrow$ as the smallest sets satisfying the following properties: *(i)* $s_0 \in \Sigma$; *(ii)* if $\langle\mathcal{I},\mathcal{M}\rangle \in \Sigma$, then for all substitutions σ for the input parameters of α and for every $\langle\mathcal{I}',\mathcal{M}'\rangle$ such that $\langle\langle\mathcal{I},\mathcal{M}\rangle,\alpha\sigma,\langle\mathcal{I}',\mathcal{M}'\rangle\rangle \in \text{EXEC}_{\mathcal{S}}$, we have $\langle\mathcal{I}',\mathcal{M}'\rangle \in \Sigma$ and $\langle\mathcal{I},\mathcal{M}\rangle \Longrightarrow \langle\mathcal{I}',\mathcal{M}'\rangle$.

Intuitively, to define the concrete transition system of the DCDS $\mathcal{S}$ we start from the initial state $s_0 = \langle\mathcal{I}_0,\emptyset\rangle$, and for each rule $Q \mapsto \alpha$ in $\mathcal{P}$, we evaluate Q over $\mathcal{I}_0$, and calculate all states s such that $\langle s_0,\alpha\sigma,s\rangle \in \text{EXEC}_{\mathcal{S}}$. Then we repeat the same steps considering each s, and so on. The computation of successor states can be done by picking all the possible combinations of resulting values for the newly introduced service calls, then checking if the successor obtained for a combination satisfies the equality constraints, filtering it away if this is not the case. It is worth noting that when new service calls are considered, the successors can be countably infinite.

EXAMPLE 4.1. Let $\mathcal{S} = \langle\mathcal{D},\mathcal{P}\rangle$ be a DCDS with data layer $\mathcal{D} = \langle\mathcal{C},\mathcal{R},\mathcal{E},\mathcal{I}_0\rangle$ and process layer $\mathcal{P} = \langle\mathcal{F},\mathcal{A},\varrho\rangle$, where $\mathcal{F} = \{f/1, g/1\}$, $\mathcal{R} = \{Q/2, P/1, R/1\}$, $\mathcal{E} = \emptyset$, $\mathcal{I}_0 = \{P(a), Q(a,a)\}$, $\varrho = \{true \mapsto \alpha\}$, $\mathcal{A} = \{\alpha\}$, and

$$\alpha : \{Q(a,a) \wedge P(x) \rightsquigarrow \{R(x)\}, P(x) \rightsquigarrow \{P(x), Q(f(x),g(x))\}\}$$

The concrete transition system $\Upsilon_{\mathcal{S}}$ contains infinitely many successors connected to the initial state. These successors result from the assignment of each possible pair of values to $f(a)$ and $g(a)$ (see also Figure 3(a). ■

EXAMPLE 4.2. Consider a variation of the DCDS described in Example 4.1, where the data layer is equipped with an equality constraint, i.e., $\mathcal{E} = \{P(x) \wedge Q(y,z) \rightarrow x = y\}$. The resulting concrete transition system has still infinitely many successors of the initial state, but the presence of the equality constraint requires to keep only those successors in which $f(a)$ returns a (see also Figure 2(a). ■

4.2 Run-Bounded Systems

We now study the verification of DCDSs with deterministic services. In particular, we are interested in the following problem: given a DCDS $\mathcal{S}$ and a temporal property Φ, check whether $\Upsilon_{\mathcal{S}} \models \Phi$. Not surprisingly, given the expressive power of DCDS as a computation model, the verification problem is undecidable for all the μ-calculus variants introduced in Section 3. In fact, we can show an even stronger undecidability result, for a very small fragment of propositional linear temporal logic (LTL) [34], namely the safety properties of the form Gp where p is propositional.

THEOREM 4.1. *There exists a DCDS $\mathcal{S}$ with deterministic services, and a propositional LTL safety property Φ, such that checking $\Upsilon_{\mathcal{S}} \models \Phi$ is undecidable.*

In the following, we isolate a notable class of DCDS for which verification of $\mu\mathcal{L}_A$ is not only decidable, but can also be reduced to standard model checking techniques.

Consider a transition system $\Upsilon = \langle\Delta,\mathcal{R},\Sigma,s_0,db,\Rightarrow\rangle$. A *run* τ in Υ is a (finite or infinite) sequence of states $s_0 s_1 s_2 \cdots$ rooted at s_0, where $s_i \Rightarrow s_{i+1}$. We use $\tau(i)$ to denote s_i and $\tau[i]$ to represent the finite prefix $s_0 \cdots s_i$ of τ. A run $\tau = s_0 s_1 s_2 \cdots$ is *(data) bounded* if the number of values mentioned inside its databases is bounded, i.e., there exists a finite bound b such that $|\bigcup_{s \text{ state of } \tau} \text{ADOM}(db(s))| < b$. This is equivalent to saying that, for every finite prefix $\tau[i]$ of τ, $|\bigcup_{j\in\{0,\ldots,i\}} \text{ADOM}(db(s_j))| < b$. We say that Υ is *run-bounded* if there exists a bound b such that every run in Υ is (data) bounded by b. A DCDS $\mathcal{S}$ is run-bounded if its concrete transition system $\Upsilon_{\mathcal{S}}$ is run-bounded.

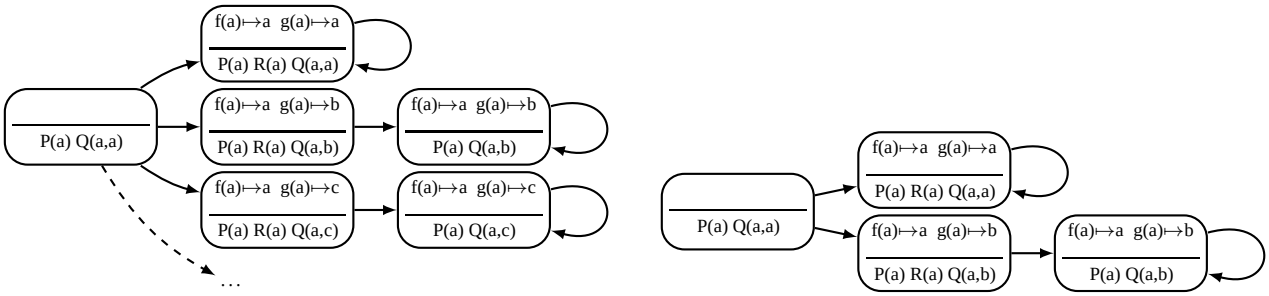
Intuitively, a (data) unbounded run represents an execution of the DCDS in which infinitely many distinct values occur because infinitely many different service calls are issued. Since we model deterministic services whose number is finite, this can only happen if some service is repeatedly called with arguments that are the result of previous service calls. This means that the values of the run indirectly depend on arbitrarily many states in the past.

Notice that run boundedness does not impose any restriction about the branching of the transition system; in particular, $\Upsilon_{\mathcal{S}}$ is typically infinite-branching because new service calls may return any possible value. We show that this restriction guarantees decidability for $\mu\mathcal{L}_A$ verification of run-bounded DCDSs with deterministic services.

THEOREM 4.2. *Verification of $\mu\mathcal{L}_A$ properties on run-bounded DCDSs with deterministic services is decidable.*

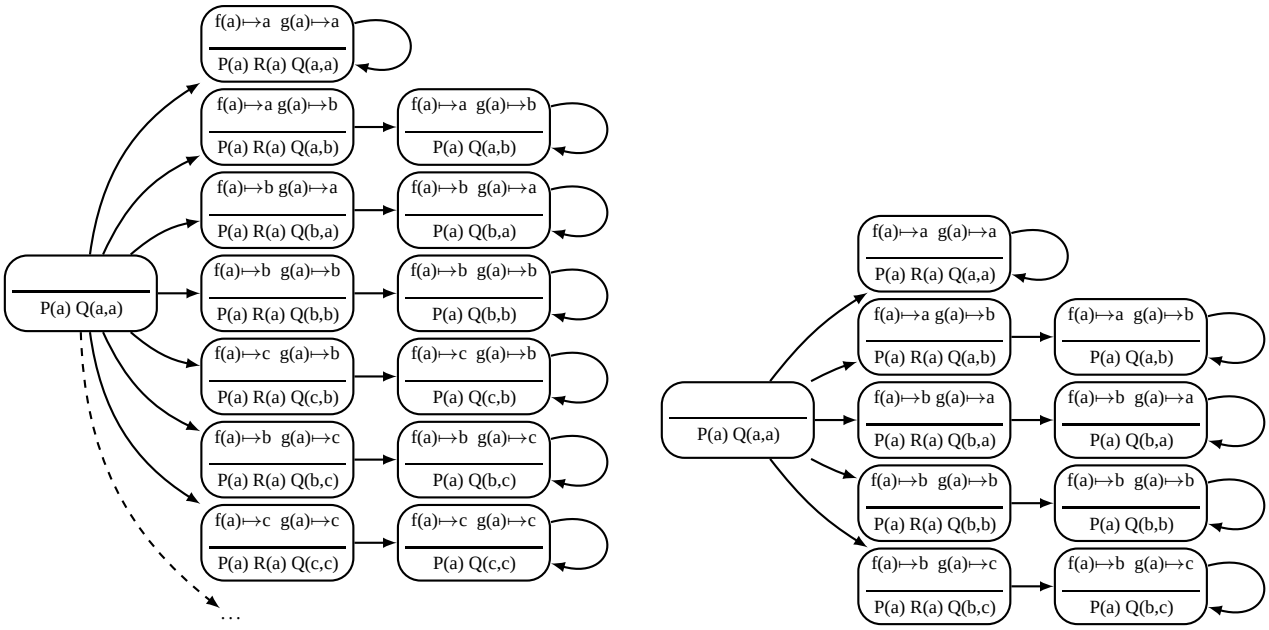
We get this result by showing that for run-bounded DCDSs there always exists an abstract finite-state transition system that is history preserving bisimilar to the concrete one, and hence satisfies the same $\mu\mathcal{L}_A$ formulae as the concrete transition system.

THEOREM 4.3. *For every run-bounded DCDS $\mathcal{S}$ with deterministic services, given its concrete transition system $\Upsilon_{\mathcal{S}}$ there exists an (abstract)* finite-state *transition system $\Theta_{\mathcal{S}}$ such that $\Theta_{\mathcal{S}}$ is history preserving bisimilar to $\Upsilon_{\mathcal{S}}$, i.e., $\Theta_{\mathcal{S}} \approx \Upsilon_{\mathcal{S}}$.*



(a) Concrete transition system (b) Abstract transition system

Figure 2: Concrete and abstract transition systems of the DCDS with deterministic services described in Example 4.2; special relations that store the service calls results are represented using a $call \mapsto value$ notation



(a) Concrete transition system (b) Abstract transition system

Figure 3: Concrete and abstract transition systems of the DCDS with deterministic services described in Example 4.1; special relations that store the service calls results are represented using a $call \mapsto value$ notation

Let Σ be the set of states of $\Theta_{\mathcal{S}}$ and $\textsc{adom}(\Theta_{\mathcal{S}}) = \bigcup_{s_i \in \Sigma} \textsc{adom}(db(s_i))$. If $\Theta_{\mathcal{S}}$ is finite-state, then there exists a bound b such that $|\textsc{adom}(\Theta_{\mathcal{S}})| < b$. Consequently, it is possible to transform a $\mu\mathcal{L}_A$ property Φ into an equivalent *finite* propositional μ-calculus formula $\textsc{prop}(\Phi)$, where $\textsc{prop}(\Phi)$ is inductively defined over the structure of Φ as the identity, except for the following case: $\textsc{prop}(\exists x.\textsc{live}(x) \wedge \Psi(x)) = \bigvee_{t_i \in \textsc{adom}(\mathcal{S})} \textsc{live}(t_i) \wedge \textsc{prop}(\Psi(t_i))$. Clearly, $\Theta_{\mathcal{S}} \models \Phi$ if and only if $\Theta_{\mathcal{S}} \models \textsc{prop}(\Phi)$.

THEOREM 4.4. *Verification of $\mu\mathcal{L}_A$ properties for run-bounded DCDSs with deterministic services can be reduced to conventional model checking of propositional μ-calculus over a finite transition system.*

By the above theorem, and recalling that model checking of propositional μ-calculus formulae over finite transition systems is decidable [22], we get Theorem 4.2.

We conclude the Section by observing that the approach presented above for $\mu\mathcal{L}_A$ does not extend to full $\mu\mathcal{L}$.

THEOREM 4.5. *There exists a DCDS $\mathcal{S}$ for which it is impossible to find a faithful finite-state abstraction that satisfies the same $\mu\mathcal{L}$ properties as $\mathcal{S}$.*

The Theorem 4.5 is proved by exhibiting, for every n, a $\mu\mathcal{L}$ property that requires the existence of at least n objects in the transition system.

Even if this observation does not imply undecidability of model checking $\mu\mathcal{L}$ properties over run-bounded DCDSs, it shows that there is no hope of reducing this problem to standard, finite-state model checking.

4.3 Weakly Acyclic DCDSs

The results presented in Section 4.2 rely on the hypothesis that the DCDS under study is run-bounded, which is a semantic restriction. A natural question is whether it is possible to check run-

boundedness of a DCDS. We provide a negative answer to this question.

THEOREM 4.6. *Checking run-boundedness of DCDSs with deterministic services is undecidable.*

To mitigate this issue, we investigate a sufficient syntactic condition that can be effectively tested over the process layer of the DCDS: if the condition is met, then the DCDS is guaranteed to be run-bounded, otherwise nothing can be said. To this end, we recast the approach of [3] in the more abstract and expressive framework here presented. In particular, we first introduce the "positive approximate" of a DCDS, which abstracts away some of its aspects. We do so for convenience, but we note that the definition of weak-acyclicity as well as our results can be stated directly over the original DCDS (in fact, we do so in condensed presentations of this work). Technically, given a DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, its *positive approximate* $\mathcal{S}^+$ is a DCDS $\langle \mathcal{D}^+, \mathcal{P}^+ \rangle$, where $\mathcal{D}^+ = \langle \mathcal{C}, \mathcal{R}, \emptyset, \mathcal{I}_0 \rangle$ corresponds to $\mathcal{D}$ without equality constraints, while $\mathcal{P}^+ = \langle \mathcal{F}, \mathcal{A}^+, \varrho^+ \rangle$ is a process layer whose actions $\mathcal{A}^+$ and process ϱ^+ are obtained as follows:

- Each condition-action rule $Q \mapsto \alpha$ in ϱ becomes $\mathsf{true} \mapsto \alpha^+$ in ϱ^+. Therefore, ϱ^+ is a process that supports the execution of every action in $\mathcal{A}^+$ at each step.
- Each action $\alpha(p_1, \ldots, p_n) : \{e_1, \ldots, e_m\}$ in $\mathcal{A}$ becomes $\alpha^+ : \{e_1^+, \ldots, e_m^+\}$ in $\mathcal{A}^+$, where each $e_i = q_i^+ \wedge Q_i^- \rightsquigarrow E_i$ becomes in turn $e_i^+ = q_i^+ \rightsquigarrow E_i$. Intuitively, the positive approximate action is obtained from the original action by removing all the parameters from its signature, and by removing all "negative" components from the query used to instantiate its effect specifications; note that the variables of q_i^+ that were parameters in α are now free variables in α^+.

The positive approximate fulfils the following key property.

LEMMA 4.1. *Given a DCDS $\mathcal{S}$, if its positive approximate $\mathcal{S}^+$ is run-bounded, then $\mathcal{S}$ is run-bounded as well.*

To derive a sufficient condition for $\mathcal{S}^+$ to be run-bounded, we can exploit a strict correspondence between the execution of an action in $\mathcal{P}^+$ and a step in the chase of a set of tuple generating dependencies (TGDs) in data exchange[2, 23]. In particular, we resort to a well-known result in data exchange, namely chase termination for *weakly acyclic* TGDs [23].[8]

In our setting, the weak acyclicity of a process layer is a property over a dataflow graph constructed by analyzing the corresponding positive approximate process layer. A non-weakly acyclic DCDS contains a service that may be repeatedly called, every time using fresh values that are directly or indirectly obtained by manipulating previous results produced by the same service. This self-dependency can potentially lead to an infinite number of calls of the same service along an execution of the system, thus making it impossible to put a bound on the data used throughout the run (see also Example 4.3). Weak acyclicity rules out such self dependencies and is actually a sufficient condition for run-boundedness.

Given a DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ with positive approximate $\mathcal{S}^+ = \langle \mathcal{D}^+, \mathcal{P}^+ \rangle$, the *dependency graph* of $\mathcal{P}^+$ is an edge-labeled directed graph $\langle N, E \rangle$ where: *(i)* $N \subseteq \mathcal{R} \times \mathbb{N}^+$ is a set of nodes such that $\langle R, i \rangle \in N$ for every $R/n \in \mathcal{R}$ and every $i \in \{1, \ldots, n\}$; *(ii)* $E \subseteq N \times N \times \{\mathsf{true}, \mathsf{false}\}$ is a set of labeled edges where

- an ordinary edge $\langle \langle R_1, j \rangle, \langle R_2, k \rangle, \mathsf{false} \rangle \in E$ if there exists an action $\alpha^+ \in \mathcal{A}^+$, an effect $q_i^+ \rightsquigarrow E_i \in \text{EFFECT}(\alpha^+)$ and a variable x such that $R_1(\ldots, t_{j-1}, x, t_{j+1}, \ldots)$ occurs in q_i^+ and $R_2(\ldots, t'_{k-1}, x, t'_{k+i}, \ldots)$ occurs in E_i;
- a special edge $\langle \langle R_1, j \rangle, \langle R_2, k \rangle, \mathsf{true} \rangle \in E$ if there exists an action $\alpha^+ \in \mathcal{A}^+$, an effect $q_i^+ \rightsquigarrow E_i \in \text{EFFECT}(\alpha^+)$ and a variable x such that $R_1(\ldots, t_{j-1}, x, t_{j+1}, \ldots)$ occurs in q_i^+, $R_2(\ldots, t'_{k-1}, t, t'_{k+i}, \ldots)$ occurs in E_i, and $t = f(\ldots, x, \ldots)$, with $f \in \mathcal{F}$.

$\mathcal{P}$ is *weakly acyclic* if the dependency graph of its approximate $\mathcal{P}^+$ does not contain any cycle going through a special edge. We say that a DCDS is weakly acyclic if its process layer is weakly acyclic (e.g., see Figure 5(a)).

Intuitively, ordinary edges represent the possible propagation (copy) of a value across states: $\langle \langle R_1, j \rangle, \langle R_2, k \rangle, \mathsf{false} \rangle \in E$ reflects the possibility that the value currently stored inside the j-th component of an R_1 tuple will be moved to the k-th component of an R_2 tuple in the next state. Contrariwise, special edges represent that a value can be taken as parameter of a service call, thus contributing to the creation of (possibly new) values across states: $\langle \langle R_1, j \rangle, \langle R_2, k \rangle, \mathsf{true} \rangle \in E$ means that the value currently stored inside the j-th component of an R_1 tuple could be used as parameter for a service call, whose result is then stored inside the k-th component of an R_2 tuple.

A cycle going through a special edge, forbidden by the weak acyclicity condition, represents that a service may be repeatedly called, every time using fresh values that are indirectly or directly obtained by manipulating previous results produced by the same service. This self-dependency can potentially lead to an infinite number of calls of the same service along an execution of the system, thus making it impossible to put a bound on the data used throughout the run.

EXAMPLE 4.3. Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \emptyset, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, where $\mathcal{F} = \{f/1\}$, $\mathcal{R} = \{R/1, Q/1\}$, $\mathcal{I}_0 = \{R(a)\}$, $\varrho = \{true \mapsto \alpha\}$ and $\mathcal{A} = \{\alpha\}$, where $\alpha : \{R(x) \rightsquigarrow Q(f(x)),\ Q(x) \rightsquigarrow R(x)\}$.

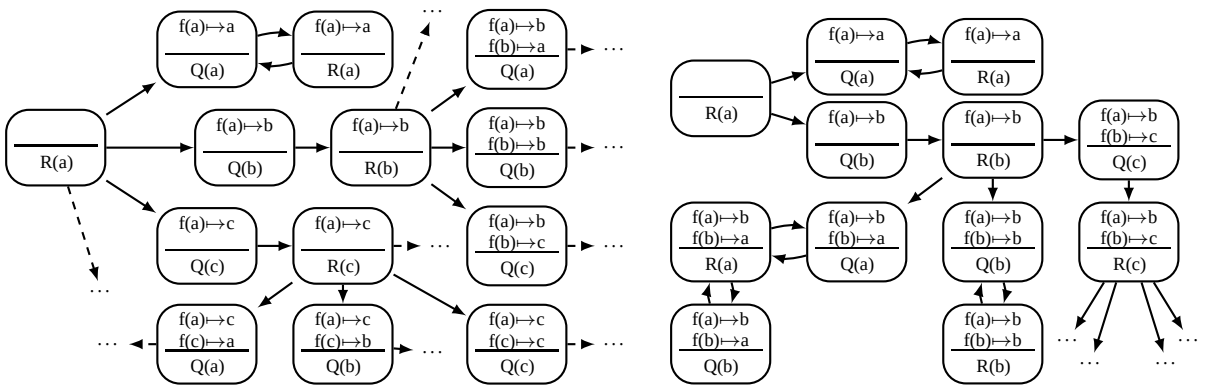
$\mathcal{S}$ is not weakly acyclic, due to the mutual dependency between R and Q that involves a call to service f. This can be easily seen from the dataflow graph (shown in Figure 5(b)), which contains a special edge from $\langle R, 1 \rangle$ to $\langle Q, 1 \rangle$, and a normal edge from $\langle Q, 1 \rangle$ to $\langle R, 1 \rangle$. Notice that, in this case, the positive approximate of $\mathcal{S}$ coincides with $\mathcal{S}$ itself. Starting from the initial state, α calls $f(a)$ and stores the result inside Q. A second execution of α transfers the result of $f(a)$ into R. When α is executed for the third time, f is called again, but using as parameter the previously obtained result. Consequently, f may return a new, fresh result, because $f(f(a))$ may be different from $f(a)$. This chain can be repeated forever, leading to possibly generate infinitely many distinct values along the run. The existence of a run in which a, $f(a)$, $f(f(a))$, $f(f(f(a)))$, ..., are all distinct values, makes it impossible to obtain a finite-state abstraction for $\mathcal{S}$ (see Figure 4(b)). ■

THEOREM 4.7. *Every weakly acyclic DCDS with deterministic services is run-bounded.*

Checking weak acyclicity is polynomial in the size of the DCDS. Thus it gives us an effective way to verify DCDSs.

THEOREM 4.8. *Verification of $\mu\mathcal{L}_A$ properties for weakly acyclic DCDSs with deterministic services is decidable, and can be reduced to model checking of propositional μ-calculus over a finite transition system.*

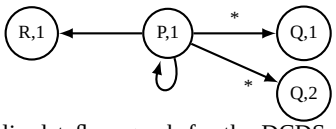
[8] Notice that using other variants of weak acyclicity is also possible [30].



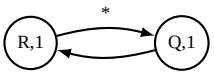
(a) Concrete transition system

(b) Abstract transition system

Figure 4: Concrete and abstract transition systems of the run-unbounded DCDS with deterministic services $\mathcal{S}$ described in Example 4.3; special relations that store the service calls results are represented using a $call \mapsto value$ notation



(a) Weakly acyclic dataflow graph for the DCDSs of Example 4.1 and 4.2



(b) Non weakly acyclic dataflow graph for the DCDS of Example 4.3

Figure 5: Examples of dataflow graphs for DCDSs with deterministic services; special edges are decorated with *

EXAMPLE 4.4. Consider the DCDSs described in Example 4.1 and 4.2. They have the same dataflow graph, which is weakly acyclic (see Figure 5(a). This guarantees that they are run-bounded and that it is possible to find a faithful finite-state abstraction from them. Two such abstractions are respectively shown in Figure 3(b) and 2(b). ■

5. NONDETERMINISTIC SERVICES

We now consider DCDSs under the assumption that services behave nondeterministically, i.e., two calls of a service with the same arguments may return distinct results during the same run. This case captures both services that model a truly nondeterministic process (e.g., human operators, random processes), and services that model stateful servers. In the remainder of this section, whenever we refer to a DCDS, services are implicitly assumed nondeterministic.

5.1 Semantics

As in the case of deterministic services, we define the semantics of a DCDS $\mathcal{S}$ in terms of a (possibly infinite) transition system $\Upsilon_{\mathcal{S}}$.

Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$. A *state* is simply a relational instance of $\mathcal{R}$ over $\mathcal{C}$ satisfying each constraint in $\mathcal{E}$. We denote the *initial state* with $\mathcal{I}_0$.

Next, we define the semantics of action application. Let α be an action in $\mathcal{A}$ of the form $\alpha(p_1, \ldots, p_m) : \{e_1, \ldots, e_m\}$ with effects $e_i = q_i^+ \wedge Q_i^- \rightsquigarrow E_i$. The parameters for α are guarded by the condition-action rule $Q \mapsto \alpha \in \varrho$. Let σ be a legal substitution for the input parameters $p_1, \ldots, p_m$ with values taken from $\mathcal{C}$.

We reuse the definition of $\text{DO}(\mathcal{I}, \alpha\sigma)$ from Section 4.1, as the instance obtained by evaluating the effects of α on instance $\mathcal{I}$. Recall that DO() generates an instance over values from the domain $\mathcal{C}$ but also over Skolem terms, which model service calls. For any such instance $\bar{\mathcal{I}}$, we denote with $\text{CALLS}(\bar{\mathcal{I}})$ the set of calls it contains. For a given set $D \subseteq \mathcal{C}$, we denote with $\text{EVALS}_D(\mathcal{I}, \alpha, \sigma)$ the set of substitutions that replace all service calls in $\text{DO}(\mathcal{I}, \alpha, \sigma)$ with values in D,

$$\text{EVALS}_D(\mathcal{I}, \alpha, \sigma) = \{\theta \mid \theta \text{ is a total function } \theta : \text{CALLS}(\text{DO}(\mathcal{I}, \alpha, \sigma)) \to D\}.$$

Each substitution in $\text{EVALS}_D(\mathcal{I}, \alpha, \sigma)$ models the simultaneous evaluation of all service calls, which replaces the calls with results selected nondeterministically from D. In the following, we refer to these substitutions as *evaluations*.

Concrete action execution. We introduce a transition relation $\text{N-EXEC}_{\mathcal{S}}$ between states, called *concrete execution* of $\alpha\sigma\theta$, such that $\langle \mathcal{I}, \alpha\sigma\theta, \mathcal{I}' \rangle \in \text{N-EXEC}_{\mathcal{S}}$ if the following holds:

1. σ is a legal parameter assignment for α in state $\mathcal{I}$,
2. $\theta \in \text{EVALS}_{\mathcal{C}}(\mathcal{I}, \alpha, \sigma)$,
3. $\mathcal{I}' = \text{DO}(\mathcal{I}, \alpha, \sigma)\theta$, and
4. $\mathcal{I}'$ satisfies the constraints $\mathcal{E}$.

Notice that, in contrast to the deterministic services case, the choice of evaluation θ is not subject to the requirement that it evaluates a service call to the same result *across* concrete execution steps. However, notice that *within* a concrete execution step, all occurrences of the same service call evaluate to the same result (modeling the fact that a call with given arguments is invoked only once per transition, and the returned result is copied as needed).

Concrete transition system. The *concrete transition system* $\Upsilon_{\mathcal{S}}$ for $\mathcal{S}$ is a transition system whose states are labeled by databases. More precisely,
$\Upsilon_{\mathcal{S}} = \langle \mathcal{C}, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$ where $s_0 = \mathcal{I}_0$ and db is such that $db(\mathcal{I}) = \mathcal{I}$. Σ and $\Rightarrow$ are defined by simultaneous induction as

the smallest sets satisfying the following properties: *(i)* $\mathcal{I}_0 \in \Sigma$; *(ii)* if $\mathcal{I} \in \Sigma$, then for all α, σ, θ and $\mathcal{I}'$ such that $\langle \mathcal{I}, \alpha\sigma\theta, \mathcal{I}' \rangle \in$ N-EXEC$_\mathcal{S}$, we have that $\mathcal{I}' \in \Sigma$, and $\mathcal{I} \Rightarrow \mathcal{I}'$.

5.2 State-Bounded Systems

We consider the verification problem for DCDS with nondeterministic services. As in the deterministic case, restrictions on both the processes and the properties are required, motivated by the following undecidability result.

THEOREM 5.1. *There exists a DCDS $\mathcal{S}$ with nondeterministic services, and a propositional LTL safety property Φ, such that checking $\Upsilon_\mathcal{S} \models \Phi$ is undecidable.*

State-bounded DCDS. Since we are interested in verifying more expressive temporal properties, we need to consider restricted classes of DCDS. We observe first that, with nondeterministic services, the run-boundedness restriction of Section 4.2 is very limiting on the form of the DCDS, as it boils down to imposing a bound on how many times each service may be called with the same arguments. Observe that this was not the case for deterministic services, where the unlimited same-argument calls are allowed, as they all return the same result. We propose a less restrictive alternative. We say that DCDS $\mathcal{S}$ is *state-bounded* if there is a finite bound b such that for each state $\mathcal{I}$ of $\Upsilon_\mathcal{S}$, $|\text{ADOM}(\mathcal{I})| < b$. Notice that, in contrast to the notion of run-boundedness, state-boundedness does allow runs in which infinitely many distinct values occur because infinitely many service calls are issued. The unboundedly many call results are distributed *across* states of the run, but may not accumulate *within* a single state. The following result shows that we also need to restrict the logic, as the one used in the deterministic case is too expressive for decidability.

THEOREM 5.2. *Verification of $\mu\mathcal{L}_A$ properties on state-bounded DCDSs with nondeterministic services is undecidable.*

We therefore restrict the property class to the logic $\mu\mathcal{L}_P \subset \mu\mathcal{L}_A$ presented in Section 3.2.

THEOREM 5.3. *Verification of $\mu\mathcal{L}_P$ properties by state-bounded DCDS with nondeterministic services is decidable.*

5.3 Abstract Transition System

We relegate the proof of Theorem 5.3 to Appendix C.3, but provide the main ideas here.

Given a DCDS $\mathcal{S}$, we show that if concrete transition system $\Upsilon_\mathcal{S}$ is state-bounded, then there is a finite-state abstract transition system $\Theta_\mathcal{S}$ whose states and edges are subsets of those in $\Upsilon_\mathcal{S}$, such that $\Theta_\mathcal{S}$ is persistence-preserving bisimilar to $\Upsilon_\mathcal{S}$ (and hence satisfies the same $\mu\mathcal{L}_P$ properties, by Theorem 3.2). Since $\Theta_\mathcal{S}$ is finite-state, the verification of $\mu\mathcal{L}_P$ properties on $\Upsilon_\mathcal{S}$ reduces to finite-state model checking on $\Theta_\mathcal{S}$, and hence is decidable.

The existence of $\Theta_\mathcal{S}$ follows from the key fact that if two states of $\Upsilon_\mathcal{S}$ are isomorphic, then they are persistence-preserving bisimilar. This implies that one can construct a finitely-branching transition system $\Theta_\mathcal{S}$ (i.e. with finite number of successors per state), such that $\Theta_\mathcal{S}$ is persistence-preserving bisimilar to $\Upsilon_\mathcal{S}$, by dropping sibling states from $\Upsilon_\mathcal{S}$ as follows: instead of listing among the successors of s one state for each possible instantiation of the service call results, just keep a *representative* state for each isomorphism type. Since the number of service calls made in each state is finite, the number of distinct isomorphism types is finite, so the finite branching follows. We call a transition system $\Theta_\mathcal{S}$ obtained as above a *pruning of* $\Upsilon_\mathcal{S}$.

Notice that despite being finitely-branching, any pruning $\Theta_\mathcal{S}$ can still have infinitely many states, as it may contain infinitely long simple runs[9] τ, along which the service calls return in each state "fresh" values, i.e., values distinct from all values appearing in the predecessors of this state on τ. This problem is solved by judiciously selecting which representatives to keep in $\Theta_\mathcal{S}$ for the successors of a state s. Namely, whenever the representatives of a given isomorphism type $\mathcal{T}$ include states generated exclusively by service calls that "recycle" values, select only such states (finitely many thereof, of course). By recycled values we mean values appearing on a path leading into s.

If $\Upsilon_\mathcal{S}$ is state-bounded, then the number of service calls per state is bounded, and due to the construction's preference for recycling, it follows that all simple runs in $\Theta_\mathcal{S}$ must have finite length. Together with the finite branching, this implies finiteness of $\Theta_\mathcal{S}$.

Notice that proving the existence of $\Theta_\mathcal{S}$ does not suffice for decidability, as the proof is non-constructive. We therefore provide an algorithm for constructing $\Theta_\mathcal{S}$ (Algorithm RCYCL). One of the technical problems we need to overcome in developing the algorithm is that we evidently cannot start from the infinite-state concrete transition system, instead exploring a portion thereof. This means that it is not obvious how to decide whether the successors of a state are generated by recycling service calls, since these calls may recycle from paths that RCYCL hasn't explored yet. Therefore, RCYCL may sometimes select non-recycling service calls even when a recycling alternative exists. However, we can prove that RCYCL constructs what we call an *eventually recycling pruning*, which in essence means it may fail to detect recycling service calls, but only a bounded number of times.

We formalize the above discussion in Appendix C.3, where we prove the following result:

THEOREM 5.4. *If input DCDS $\mathcal{S}$ is state-bounded, then every possible run of Algorithm* RCYCL *terminates, yielding a finite eventually recycling pruning $\Theta_\mathcal{S}$ of $\Upsilon_\mathcal{S}$, with $\Upsilon_\mathcal{S} \sim \Theta_\mathcal{S}$.*

Theorem 5.4 and Theorem 3.2 directly imply Theorem 5.3.

Figures 7 and 6 illustrate two concrete transition systems, and possible recycling prunings for them.

5.4 GR-Acyclic DCDSs

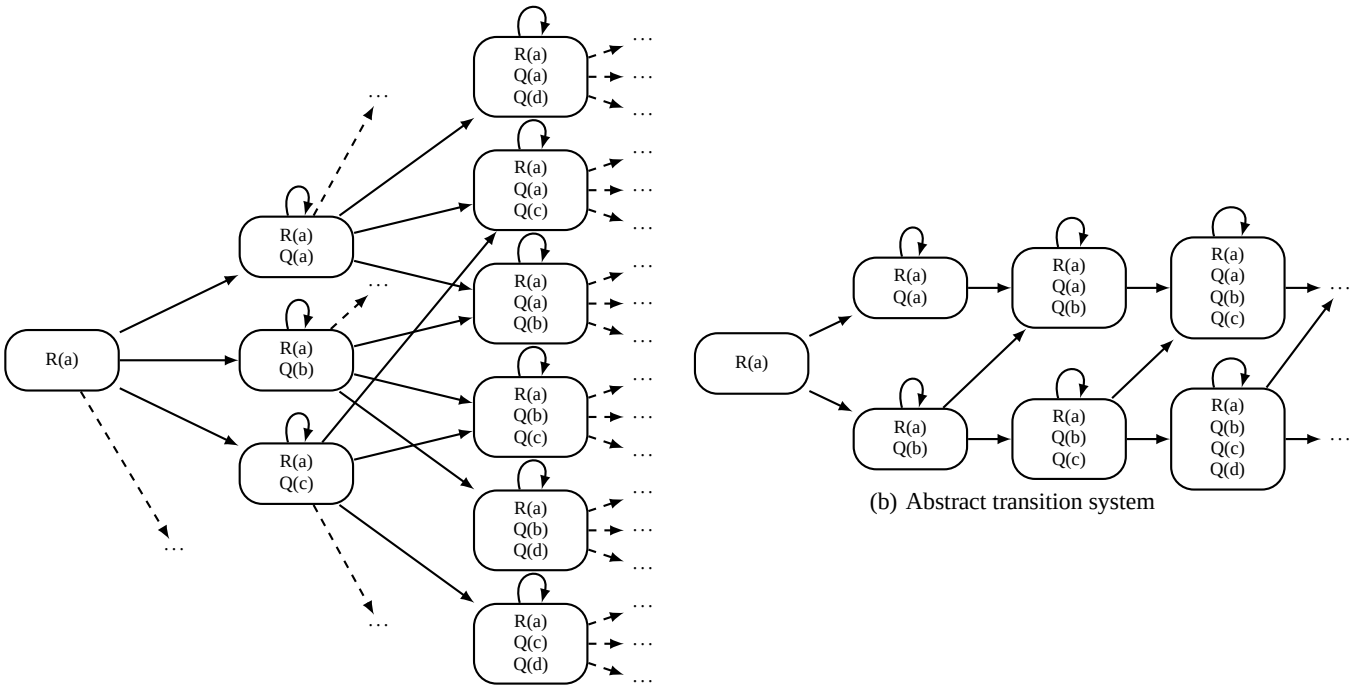
As with run-boundedness in the deterministic services case, for nondeterministic services the state-boundedness restriction is a semantic property. We investigate whether it can be effectively checked.

THEOREM 5.5. *Checking state-boundedness of DCDSs is undecidable.*

Consequently we propose a sufficient syntactic restriction.

Intuitively, for a run to have unbounded states, it must issue unboundedly many service calls. Since there are only a bounded number of effects in the process layer specification, there must exist some service-calling effect that "cyclically generates" fresh values (i.e. is invoked infinitely many times during the run). Notice that unbounded generation of fresh values is insufficient for state-unboundedness: these values must also accumulate in the states. But by definition of the DCDS semantics, a transition drops ("forgets") all values that are not explicitly copied ("recalled") into the successor. Therefore, to accumulate, a value must be "cyclically recalled" througout the run (it must be copied infinitely many times from relation to relation).

[9]We call a run *simple* if no state appears more than once in the run.



(a) Concrete transition system

(b) Abstract transition system

Figure 6: Concrete and abstract transition systems of the state-unbounded DCDS with nondeterministic services of Example 5.2.

GR-acyclicity is stated in terms of a dataflow graph constructed by analyzing the process layer. The graph identifies how service calls and value recalls can chain. In essence, GR-acyclicity requires the absence of a "generate cycle" that feeds into a "recall cycle".

GR-acycliclicity. Let $\mathcal{A}$ be a set of actions, and $\mathcal{A}^+$ its positive approximate (Section 4.3). We call *dataflow graph* of $\mathcal{A}$ the directed edge-labeled graph $\langle N, E\rangle$ whose set N of nodes is the set of relation names occurring in $\mathcal{A}$, and in which each edge in E is a 4-tuple (R_1, id, R_2, b), where R_1 and R_2 are two nodes in N, id is a (unique) edge identifier, and b is a boolean flag used to mark *special* edges. Formally, E is the minimal set satisfing the following condition: for each effect e of $\mathcal{A}^+$, each $R(t_1, \ldots, t_m)$ in the body of e, each $Q(t'_1, \ldots, t'_{m'})$ in the head of e, and each $i \in \{1, \ldots, m'\}$:

- if t'_i is either an element of $\textsc{adom}(\mathcal{I}_0)$ or a free variable, then $(R, id, Q, \mathsf{false}) \in E$, where id is a fresh edge identifier.
- if t'_i is a service call, then $(R, id, Q, \mathsf{true}) \in E$, where id is a fresh edge identifier.

We say that $\mathcal{A}$ is *GR-acyclic* if there is no path $\pi = \pi_1\pi_2\pi_3$ in the dataflow graph of $\mathcal{A}$, such that π_1, π_3 are simple cycles and π_2 is a path containing a special edge that is disjoint from the edges of π_1. We say that a process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho\rangle$ is GR-acyclic, if $\mathcal{A}$ is GR-acyclic. We call a DCDS GR-acyclic if its process layer is GR-acyclic.

Notice that GR-acyclicity is a purely syntactic notion. Moreover, it can be checked in PTIME since the dataflow graph has size polynomial in the size of the process layer specification.

THEOREM 5.6. *Any GR-acyclic DCDS is state-bounded.*

We show the proof in Appendix C.4 but provide some intuition here, noting that the dataflow analysis is significantly more subtle than suggested above.

First, note that ordinary edges correspond to an effect copying a value from a relation of the current state to a relation of the successor state. Special edges correspond to feeding a value of the current state to a service call and storing the result in a relation of the successor state. Note that the cycles π_1 and π_3 allow both kinds of edges, reflecting the insight that the size of the state is affected in the same way regardless of whether a value is *copied* to the successor, or it is *replaced* with a service call result (see Example 5.2 and Example 5.3 for illustrations of state-unboundedness arising from each case). π_1, π_3 are both "recall cycles": the number of values moving around them does not decrease (this is of course a conservative statement; reality depends on the semantics of queries in the effects, which is abstracted away). Note that π_2 contains a special edge E, which means that the values moving around π_1 are cyclically fed into the service call f of E. The key insight here is that, even if the set of values moving around π_1 does not change (no special edges in π_1 replace them), and thus the service call f sees the same bounded set of distinct arguments over time, it can still generate an unbounded number of fresh values because f is nondeterministic. $\pi_1\pi_2$ constitute the "generate cycle" we mention above. The generated values are stored in the recall cycle π_3, where they accumulate and force the size of the relations of π_3 to grow unboundedly.

EXAMPLE 5.1. Let us consider again the DCDS $\mathcal{S}$ described in Example 4.3, this time considering $f/1$ as a nondeterministic service. The resulting concrete transition system is shown in Figure 7(a). Even if $\mathcal{S}$ is not run-bounded, it is state-bounded, because in every state its database consists of only one tuple. This is attested by the dataflow graph shown in Figure 7(a), and guarantees the ex-

istence of a faithful finite-state abstraction. One such finite-state abstraction is reported in Figure 7(b). ■

EXAMPLE 5.2. Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \emptyset, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, where $\mathcal{F} = \{f/1\}$, $\mathcal{R} = \{R/1, Q/1\}$, $\mathcal{I}_0 = \{R(a)\}$, $\varrho = \{true \mapsto \alpha\}$, $\mathcal{A} = \{\alpha\}$ and $\alpha : \{R(x) \rightsquigarrow R(x), R(x) \rightsquigarrow Q(f(x)), Q(x) \rightsquigarrow Q(x)\}$.

$\mathcal{S}$ is not GR-acyclic, because each R tuple is continuously copied, and at the same time continuously issues a call to service f that is then stored into a Q tuple, which is continuously copied as well. This is attested by the dataflow graph of Figure 8(b)).

The overall effect caused by the iterated application of α is that fresh values are continuously generated and accumulated, making $\mathcal{S}$ state-unbounded. Consider for example the application of action α in state $\mathcal{I}_0$. It leads to an infinite number of successors, each one of the form $\{R(a), Q(v)\}$ where v is the value returned by $f(a)$. Consider now a second application of α in one of these states. It again leads to an infinite number of successors, due to the nondeterminism of $f(a)$. In particular, each successor has the form $\{R(a), Q(v), Q(v')\}$, where v' is the result of the second call $f(a)$. When $v' \neq v$, the number of tuples is increased from 2 to 3. By executing α over and over again, for some successors the value returned by a new call $f(a)$ will be distinct from all the ones already stored in Q. This causes an indefinite increment of the database size due to the continuous insertion of fresh Q tuples. Such behavior is clearly shown in the concrete transition system of $\mathcal{S}$, depicted in Figure 6(a). Figure 6(b) shows instead one possible corresponding abstraction; even if the abstraction approach ensures that the generated transition system is finite-branching, some of its runs pass through an infinite number of distinct, growing states. ■

EXAMPLE 5.3. Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \emptyset, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, where $\mathcal{F} = \{f/1, g/1\}$, $\mathcal{R} = \{R/1\}$, $\mathcal{I}_0 = \{R(a)\}$, $\varrho = \{true \mapsto \alpha\}$, $\mathcal{A} = \{\alpha\}$ and $\alpha : \{R(x) \rightsquigarrow \{R(f(x)), R(g(x))\}\}$.

$\mathcal{S}$ is not GR-acyclic, because its dataflow graph, shown in Figure 8(c), contains a unique node R with two distinct special looping edges from R to R itself. Indeed, every time α is executed, each R tuple contained in the current database may generate two R tuples in the next state, such that each such new tuple contains a value different from all the other ones. Therefore, even if the newly generated values are not accumulated, in the "worst" case the number of R tuples is doubled every time α is executed. A sample run of the system could be the following. Starting from $\mathcal{I}_0$, α calls $f(a)$ and $g(a)$, getting b and c as result and obtaining the state $\{R(b), R(c)\}$. A second execution of α involves now 4 service calls ($f(b)$, $g(b)$, $f(c)$, $g(c)$), which may return 4 different new values, e.g. leading to state $\{R(d), R(e), R(f), R(g)\}$, and so on. ■

GR$^+$-Acyclicity. This relaxation of GR-acyclicity is based on the insight that, for a cycle $\Upsilon_\mathcal{S}$ in the dataflow graph to truly preserve the number of values moving in it, $\Upsilon_\mathcal{S}$'s edges must not all be simultaneously inactive. We say that an edge is *active* in a step of the run when some action corresponding to it executes. By the DCDS semantics, if all edges of $\Upsilon_\mathcal{S}$ are simultaneously inactive, then none of the corresponding copy/call operations are executed and all relations involved in $\Upsilon_\mathcal{S}$ forget their value in the next state. $\Upsilon_\mathcal{S}$ is effectively flushed.

GR$^+$-Acyclicity is a relaxation that does allow path $\pi = \pi_1\pi_2\pi_3$ as in the definition of GR-Acyclicity, provided that π_2 contains an edge e that cannot be active at the same time as any of the subsequent edges in $\pi_2\pi_3$.

Semantically this ensures that in order for the generate cycle $\pi_1\pi_2$ to push fresh values toward recall cycle π_3, some action corresponding to e must execute, and in the meantime all actions maintaining the values in cycle π_3 are disabled, thus flushing π_3. π_3 thus receives an unbounded number of waves of fresh values from $\pi_1\pi_2$, but it forgets each wave before the next arrives.

Of course, the property of being active at the same time is semantic in nature, but we give a sufficient syntactic condition. Associate with every edge e in the dataflow graph the set $actions(e)$ of actions it corresponds to (this set can be computed via simple inspection of the process layer). Then edges e_1, e_2 are not simultaneously active if $actions(e_1) \cap actions(e_2) = \emptyset$.

The DCDSs discussed in Example 5.2 and 5.3 are not GR$^+$-acyclic. Indeed, they are not GR-acyclic, and all the edges contained in their dataflow graphs can be simultaneously active, because they all correspond to a single action.

We observe that GR-acyclicity is not related to weak acyclicity. In particular, a DCDS may be GR-acyclic but not weakly acyclic (see Example 5.1).

As with any sufficient syntactic condition for an undecidable semantic property, an infinite succession of refinements of GR-acyclicity is possible, each relaxing the condition to allow more DCDS classes. We propose a very powerful relaxation in Appendix C.4, GR$^+$-acyclicity. Appendix E shows a full-fledged DCDS example that conforms to GR$^+$-acyclicity, showing that it admits a practically relevant DCDS class.

Theorem 5.6 and Theorem 5.3 imply:

THEOREM 5.7. *Verification of $\mu\mathcal{L}_P$ properties for GR$^+$-acyclic DCDS with nondeterministic services is decidable.*

6. DISCUSSION

Complexity. Both in the case of weakly acyclic DCDSs with deterministic services and of GR$^+$-acyclic DCDSs with nondeterministic services, our construction generates a finite transition system whose number of states is exponential in the size of the DCDS. Let Φ be a $\mu\mathcal{L}_A$ or $\mu\mathcal{L}_P$ formula of size ℓ with k alternating nested fixpoints. Then, considering the complexity or propositional μ-calculus model checking on finite transition systems [22], the complexity of verification of Φ over a DCDS of size n is $O(2^n \cdot n^\ell)^k$, hence in EXPTIME.

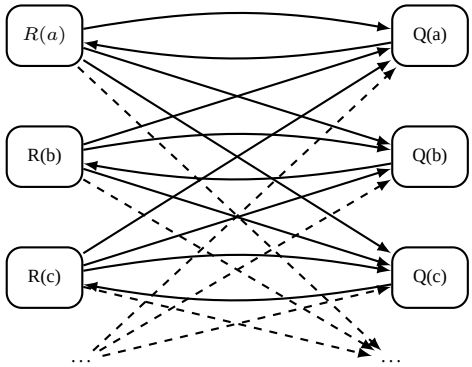
Comparison of the two semantics. It is natural to ask how the expressivities of the two DCDS flavors compare. Interestingly, we can show that for unrestricted DCDSs, the two semantics are equivalent from the point of view of expressive power, i.e. any DCDS with deterministic services can be simulated by a DCDS with nondeterministic services, and conversely. However, we show below that the two semantics are not equivalent with respect to decidability of verification.

Consider first the reduction from deterministic to nondeterministic services.

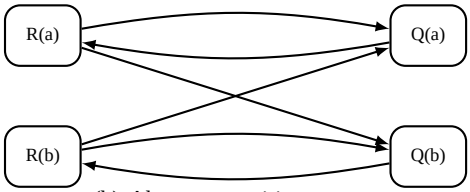
THEOREM 6.1. *Let D be a DCDS with deterministic services and schema $\mathcal{R}_D$. Then one can rewrite D in linear time to a DCDS N with nondeterministic services and schema $\mathcal{R}_N$, such that (i) $\mathcal{R}_N$ includes $\mathcal{R}_D$ and (ii) the projection of Υ_N to $\mathcal{R}_D$ coincides with Υ_D, and (iii) if D is run-bounded, then N is state-bounded.*

We turn next to the converse reduction.

THEOREM 6.2. *Let N be a DCDS with nondeterministic services and schema $\mathcal{R}_N$. Then one can rewrite N in linear time to a DCDS D with deterministic services and schema $\mathcal{R}_D$, such that (i)*



(a) Concrete transition system

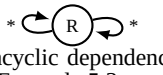


(b) Abstract transition system

Figure 7: Concrete and abstract transition systems obtained when the DCDS described in Example 4.3 has nondeterministic services

(a) GR-acyclic dataflow graph for the DCDSs of Example 4.1 and 4.2

(b) Non GR-acyclic dataflow graph for the DCDS of Example 5.2



(c) Non GR-acyclic dependency graph for the DCDS of Example 5.3

Figure 8: Examples of dependency graphs for DCDSs with nondeterministic services; special edges are decorated with *.

$\mathcal{R}_D$ includes $\mathcal{R}_N$ and (ii) the projection of Υ_D to $\mathcal{R}_N$ coincides with Υ_N.

The above reductions show that for unrestricted DCDS, deterministic and nondeterministic services are equivalent with respect to expressive power. However, they are not equivalent with respect to decidability of verification. This is because state-boundedness of the DCDS with nondeterministic services does not imply run-boundedness of the rewritten DCDS with deterministic services. In fact, one can prove that there exists no reduction from state-bounded DCDS with nondeterministic services to run-bounded DCDS with deterministic services: recall that for properties from $\mu\mathcal{L}_A - \mu\mathcal{L}_P$, verification is decidable for the latter (by Theorem 4.3), and undecidable for the former (Theorem 5.2). In particular, the reduction we use to prove Theorem 6.2 yields a non-weakly-acyclic DCDS and is therefore not pertinent to verification decidability.

In contrast, for the converse reduction of Theorem 6.1, observe that whenever D is run-bounded, N is state-bounded. Therefore, if we restrict the property language to $\mu\mathcal{L}_P$, decidability of verification for run-bounded DCDSs with deterministic services follows as a corollary of the reduction. Recall however that decidability holds even for the larger logic $\mu\mathcal{L}_A$ (by Theorem 4.2). Our proof of Theorem 4.2 exploits the reduction as well, though additional technical contribution is needed to handle $\mu\mathcal{L}_A$.

Mixed semantics. The reduction in Theorem 6.1 allows us to verify $\mu\mathcal{L}_P$ properties for DCDSs with a *mix* of deterministic and nondeterministic services, by first rewriting to a DCDS with exclusively nondeterministic services (as long as the rewritten DCDS is GR-acyclic). We give an example of a DCDS with mixed service semantics in Appendix E.

Support for arbitrary integrity constraints. We remark that, by exploiting the equality constraints, we can extend our decidability results to support integrity constraints on the database expressed as arbitrary FO sentences under the active domain semantics. First, note that the definition of DCDS semantics is independent of the type of constraints used, as it simply requires their satisfaction by each state of the concrete transition system. Now consider a DCDS $\mathcal{S}$ with an FO integrity constraint *IC* defined on its schema. We can rewrite $\mathcal{S}$ to enforce *IC* using equality constraints. To this end, we add a binary auxiliary relation *aux* to the schema, initialized in the initial state to contain the tuple $\langle a, b\rangle$ of distinct constants. We add to each action an effect that simply copies *aux* between states, ensuring the persistence of fact $aux(a, b)$ throughout the run. Finally, we add an equality constraint $ec := \neg IC \wedge aux(x, y) \rightarrow x = y$. Notice now that $\mathcal{S}$' will never execute an action that violates *IC*, because that would violate ec. Equality constraints also prove instrumental in modeling artifact systems, described next.

Connection with the artifact model. In terms of expressive capabilities, our DCDS model is equivalent to a business process model known in the literature as the *artifact model* (see Section 7). While variations thereof abound, they are virtually all special cases of the following general model. In it, given a relational schema $\mathcal{T}$, an *artifact of type* $\mathcal{T}$ (or $\mathcal{T}$-artifact) is a tuple of schema $\mathcal{T}$. The attributes of the tuple are known as *artifact variables*, and they must include an *id* attribute that uniquely identifies each artifact. An *artifact system* has a schema comprising a collection of types $\{T_i\}_{i\in\{1,\ldots,n\}}$, and the schema $\mathcal{R}_{DB}$ of an underlying relational database. The *instance* of an artifact system consists of a relation I_i for each type $\mathcal{T}_i$ and a database of schema $\mathcal{R}_{DB}$. The artifact system also has a collection of actions (usually called "services", a term we avoid here to rule out confusion with the external services of the DCDS model). The exection of an action evolves the current instance into its successor. Each action has a *pre-condition* which is a FO sentence over the artifact schema, evaluated over the current artifact instance under the active domain semantics. The pre-condition must hold for an action to be eligible to execute. Actions are also equipped with a *post-condition* which is usually an $\exists$FO formula relating the current and the successor instances (if R is a relation in the schema, the post-condition's R-atoms refer to the current instance, while

R' atoms refer to the successor). By $\exists$FO we mean existential FO logic, in which only existential quantifiers are allowed, and they must appear in the scope of an even number of negations. Existentially quantified variables are not interpreted over the active domain, but over the possible infinite domain. They model external inputs from the environment the artifact system evolves in.

While we do not show a formal reduction between the two models, we sketch here how a DCDS process can simulate an artifact-based one. The DCDS can model the sets I_i of $\mathcal{T}_i$-artifacts using an integrity constraint to enforce the uniqueness of the *id* attribute. The pre-conditions of artifact actions correspond to the conditions in the DCDS condition-action rules. Artifact post-conditions ψ can be simulated by DCDS effects, after rewriting ψ to Skolem normal form and introducing for each resulting Skolem term a nondeterministic service call. The fact that post-conditions can contain disjunction while effects are conjunctive and positive is no impediment: the additional expressivity needed can be transferred to the DCDS condition-action rules, if necessary modeling one artifact transition step with several DCDS transition steps.

7. RELATED WORK

As discussed in Section 6, the unrestricted artifact-centric and DCDS models have equivalent expressive capabilities. Our work is therefore most closely related to prior work on verification of artifact-centric business processes. The difference lies in how each work trades off between restricting the class of business processes versus the class of properties to verify.

Artifact-centric processes with no database. Work on formal analysis of artifact-based business processes in restricted contexts has been reported in [24, 25, 7]. Properties investigated include reachability [24, 25], general temporal constraints [25], and the existence of complete execution or dead end [7]. For the variants considered in each paper, verification is generally undecidable; decidability results were obtained only under rather severe restrictions, e.g., restricting all pre-conditions to be "true" [24], restricting to bounded domains [25, 7], or restricting the pre- and post-conditions to be propositional, and thus not referring to data values [25]. [15] adopts an artifact model variation with arithmetic operations but no database. It proposes a criterion for comparing the expressiveness of specifications using the notion of *dominance*, based on the input/output pairs of business processes. Decidability relies on restricting runs to bounded length. [37] addresses the problem of the existence of a run that satisfies a temporal property, for a restricted case with no database and only propositional LTL properties. All of these works model no underlying database (and hence no integrity constraints).

Artifact-centric processes with underlying database. More recently, two lines of work have considered artifact-centric processes that also model an underlying relational database. One considers branching time, one only linear time.

Branching time. Our approach stems from a line of research that has started with [16] and continued with [3] and [5] in the context of artifact-centric processes. The connection between evolution of data-centric dynamic systems and data exchange that we exploit in this paper was first devised in [16]. There the dynamic system transition relation itself is described in terms of TGDs mapping the current state to the next, and the evolution of the system is essentially a form of chase. Under suitable weak acyclicity conditions such a chase terminates, thus making the DCDS transition system finite. A first-order μ-calculus without first-order quantification across states is used as the verification formalism for which decidability is shown. Notice the role of getting new objects/values from the external environment, played here by service calls, is played there by nulls. These ideas where further developed in [3], where TGDs where replaced by action rules with the same syntax as here. Semantically however the dynamic system formalism there is deeply different: what we call here service calls are treated there as uninterpreted Skolem terms. This results in an ad-hoc interpretation of equality which sees every Skolem term as equal only to itself (as in the case of nulls [16]). The same first-order μ-calculus without first-order quantification across states of [16] is used as the verification formalism, and a form of weak acyclicity is used as a sufficient condition for getting finite-state transition systems and decidability.

In the case of deterministic services, our framework is directly inspired by [3], though here we do interpret service calls. This decision is motivated by our goal of modeling real-life external services, for which two distinct service calls may very well return equal results, even under the deterministic semantics (for instance if the same service is called with different arguments, or if distinct services are invoked). Interpreting service calls raises a major challenge: even under the run-bounded restriction, the concrete transition system is infinite, because it is infinitely branching. (a service call can be interpreted with any of the constants from the infinite domain). In contrast to [3], what we show in this case is not that the concrete transition system is finite (it never is), but that it is *bisimilar* to a finite abstract transition system. This leads to a proof technique that is interesting in its own right, being based on novel notions of bisimilarity for the considered μ-calculus variants. The reason standard bisimilarity is insufficient is that our logics $\mu\mathcal{L}_P$ and $\mu\mathcal{L}_A$ allow first-order quantification across states, so bisimilarity must respect the connection between values appearing both in the current and successor state. Our decision to include first-order quantification across states was motivated by the need to express liveness properties that refer to the same data at various points in time (e.g. "if student x is enrolled now and continues to be enrolled in the future, then x will eventually graduate").

Inspired by [3], [5] builds a similar framework where actions are specified via pre- and post-conditions given as FO formulae interpreted over active domains. The verification logic considered is a first-order variant of CTL with no quantification across states. Thus, it inherits the limitations discussed above on expressibility of liveness properties. In addition, the limited temporal expressivity of CTL precludes expressing certain desirable properties such as fairness. [5] shows that under the assumption that each state has a bounded active domain, one can construct an abstract finite transition system that can be checked instead of the original concrete transition system, which is infinite-state in general. The approach is similar to the one we developed independently for nondeterministic services, however without quantification across states, standard bisimilarity suffices. As opposed to our work, the decidability of checking state-boundedness is not investigated in [5], and no sufficient syntactic conditions are proposed.

Linear time. Publication [21] considers an artifact model that has the same expressive capabilities as an unrestricted class of DCDS in which the infinite domain is equipped with a dense linear order, which can be mentioned in pre-, post-conditions, and properties. Runs can receive unbounded external input from an infinite domain, and this input corresponds to nondeterministic services in a DCDS. Verification is decidable even if the input accumulates in states, and runs are neither run-bounded, nor state-bounded. However, this expressive power requires restrictions that render the result incomparable to ours. First, the property language is a first-order extension of LTL, and it is shown that extension to branching time (CTL*) leads to undecidability. Second, the

DETERMINISTIC SERVICES				NONDETERMINISTIC SERVICES			
	$\mu\mathcal{L}$	$\mu\mathcal{L}_A$	$\mu\mathcal{L}_P$		$\mu\mathcal{L}$	$\mu\mathcal{L}_A$	$\mu\mathcal{L}_P$
unrestricted	**U**	**U**	**U** [1]	*unrestricted*	**U**	**U**	**U** [1]
bounded-run	**?** [2]	**D** [3]	**D**	*bounded-state*	**U**	**U**	**D** [3]

[1] The result is even stronger: it holds for propositional LTL.
[2] Decidability cannot be established via a faithful finite-state abstraction.
[3] Decidability is obtained via reduction to finite-state model checking.

Table 1: Summary of our (un)decidability results.

formulae in pre-, post-conditions and properties access read-only and read-write database relations differently, querying the latter only in limited fashion. In essence, data can be arbitrarily accumulated in read-write relations, but these can be queried only by checking that they contain a given tuple of constants. It is shown that this restriction is tight, as even the ability to check emptiness of a read-write relation leads to undecidability. In addition, no integrity constraints are supported as it is shown that allowing a single functional dependency leads to undecidability. [19] disallows read-write relations entirely (only the artifact variables are writable), but this allows the extension of the decidability result to integrity constraints expressed as embedded dependencies with terminating chase, and to any decidable arithmetic. Again the result is incomparable to ours, as our modeling needs include read-write relations and their unrestricted querying.

Infinite-state systems. DCDSs are a particular case of infinite-state systems. Research on automatic verification of infinite-state systems has also focused on extending classical model checking techniques (e.g., see [14] for a survey). However, in much of this work the emphasis is on studying recursive control rather than data, which is either ignored or finitely abstracted. More recent work has been focusing specifically on data as a source of infinity. This includes augmenting recursive procedures with integer parameters [10], rewriting systems with data [9], Petri nets with data associated to tokens [28], automata and logics over infinite alphabets [12, 11, 31, 20, 27, 8, 9], and temporal logics manipulating data [20]. However, the restricted use of data and the particular properties verified have limited applicability to the business process setting we target with the DCDS model.

8. CONCLUSIONS

We summarize our results in Table 1 (arrows denote implications between results). We note that exhibiting a finite faithful abstraction of a concrete transition system is more than a means towards showing decidability, being a desirable goal in its own right as the most promising avenue towards practical implementation. Notice that we list as open the verification of $\mu\mathcal{L}$ properties on bounded-run DCDSs with deterministic services, but recall from Section 4.2 that in this case there exists no faithful finite-state abstract transition system.

We believe that DCDSs provide a natural and expressive model for business processes powered by an underlying database, and thus are an ideal vehicle for foundational research with potential to transfer to alternative models.

Note that the design space for FO extensions of propositional μ-calculus is broad, and notoriously contains bounded-state settings for which satisfiability of even modest extensions of propositional LTL is highly undecidable (e.g. LTL with the freeze quantifier over infinite data words [20]). In light of this, our decidability results come as a pleasant surprise, and the two $\mu\mathcal{L}$ variants studied here, paired with the respective DCDS classes, strike a fortuitous balance between expressivity and verification feasibility.

9. REFERENCES

[1] S. Abiteboul, P. Bourhis, A. Galland, and B. Marinoiu. The AXML artifact model. In *TIME*, 2009.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[3] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, R. De Masellis, and P. Felli. Foundations of relational artifacts verification. In *BPM*, 2011.

[4] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[5] F. Belardinelli, A. Lomuscio, and F. Patrizi. Verification of deployed artifact systems via data abstraction. In *ICSOC*, 2011.

[6] K. Bhattacharya et al. A model-driven approach to industrializing discovery processes in pharmaceutical research. *IBM Systems Journal*, 44(1), 2005.

[7] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *BPM*, 2007.

[8] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, pages 7–16, 2006.

[9] A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting systems with data. In *FCT*, 2007.

[10] A. Bouajjani, P. Habermehl, and R. Mayr. Automatic verification of recursive procedures with one integer parameter. *Theoretical Computer Science*, 295, 2003.

[11] P. Bouyer. A logical characterization of data languages. *IPL*, 84(2), 2002.

[12] P. Bouyer, A. Petit, and D. Thérien. An algebraic approach to data languages and timed languages. *Information and Computation*, 182(2), 2003.

[13] J. Bradfield and C. Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, volume 3. Elsevier, 2007.

[14] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification of infinite structures. In *Handbook of Process Algebra*. Elsevier Science, 2001.

[15] D. Calvanese, G. De Giacomo, R. Hull, and J. Su. Artifact-centric workflow dominance. In *ICSOC-ServiceWave*, 2009.

[16] P. Cangialosi, G. De Giacomo, R. De Masellis, and R. Rosati. Conjunctive artifact-centric services. In *ICSOC*, 2010.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, 1999.

[18] D. Cohn and R. Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Engineering Bullettin*, 32(3), 2009.

[19] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. In *ICDT*, 2011.

[20] S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Trans. on Computational Logic*, 10(3), 2009.

[21] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT*, 2009.

[22] E. A. Emerson. Model checking and the mu-calculus. In *Descriptive Complexity and Finite Models*, 1996.

[23] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[24] C. E. Gerede, K. Bhattacharya, and J. Su. Static analysis of business artifact-centric operational models. In *IEEE Int. Conf. on Service-Oriented Computing and Applications*, 2007.

[25] C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *ICSOC*, 2007.

[26] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM 2008 Confederated Int. Conf.*, volume 5332 of *LNCS*, 2008.

[27] M. Jurdzinski and R. Lazić. Alternation-free modal mu-calculus for data trees. In *LICS*, 2007.

[28] R. Lazić, T. Newcomb, J. Ouaknine, A. Roscoe, and J. Worrell. Nets with tokens which carry data. In *ICATPN*, 2007.

[29] D. C. Luckham, D. M. R. Park, and M. Paterson. On formalised computer programs. *J. Computer and System Sciences*, 4(3), 1970.

[30] M. Meier, M. Schmidt, F. Wei, and G. Lausen. Semantic query optimization in the presence of types. In *PODS*, pages 111–122, 2010.

[31] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. on Computational Logic*, 5(3), 2004.

[32] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3), 2003.

[33] D. M. R. Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3(2), 1976.

[34] A. Pnueli. The temporal logic of programs. In *FOCS*, 1977.

[35] C. Stirling. *Modal and Temporal Properties of Processes.* Springer, 2001.

[36] W. M. P. van der Aalst, P. Barthelmess, C. A. Ellis, and J. Wainer. Proclets: A framework for lightweight interacting workflow processes. *Int. J. Cooperative Information Systems*, 10(4), 2001.

[37] X. Zhao, J. Su, H. Yang, and Z. Qiu. Enforcing constraints on life cycles of business artifacts. In *TASE*, 2009.

APPENDIX

A. VERIFICATION

In this appendix we give the bisimulation invariance results for $\mu\mathcal{L}_A$ and $\mu\mathcal{L}_P$.

A.1 History Preserving Mu-Calculus

We prove *history preserving bisimulation invariance* for $\mu\mathcal{L}_A$. We adopt a two-step approach. We first prove the result for the logic $\mathcal{L}_A$, obtained from $\mu\mathcal{L}_A$ dropping the predicate variables and the fixpoint constructs. Such a logic corresponds to a first-order variant of the Hennessy Milner logic; note that the semantics of this logic is completely independent from the second-order valuation. We then extend the result to the whole $\mu\mathcal{L}_A$ by dealing with fixpoints.

LEMMA A.1. *Consider two transition systems* $\Upsilon_1 = \langle \Delta_1, \mathcal{R}, \Sigma_1, s_{01}, db_1, \Rightarrow_1 \rangle$ *and* $\Upsilon_2 = \langle \Delta_2, \mathcal{R}, \Sigma_2, s_{02}, db_2, \Rightarrow_2 \rangle$*, a partial bijection* h *between* Δ_1 *and* Δ_2 *and two states* $s_1 \in \Sigma_1$ *and* $s_2 \in \Sigma_2$ *such that* $s_1 \approx_h s_2$*. Then for every (open) formula* Φ *of* $\mathcal{L}_A$*, and every valuations* v_1 *and* v_2 *that assign to each of its free variables a value* $d_1 \in \text{ADOM}(db_1(s_1))$ *and* $d_2 \in \text{ADOM}(db_2(s_2))$*, such that* $d_2 = h(d_1)$*, we have that*

$$\Upsilon_1, s_1 \models \Phi v_1 \text{ if and only if } \Upsilon_2, s_2 \models \Phi v_2.$$

PROOF. We proceed by induction on the structure of Φ.

Local first-order queries (base case) Consider $\Phi = Q$, where Q is an (open) FO query. Since h induces an isomorphism between $db(s_1)$ and $db(s_2)$, for every valuations v_1 and v_2 that assign to each free variable of Q a value $d_1 \in \text{ADOM}(db_1(s_1))$ and $d_2 \in \text{ADOM}(db_2(s_2))$, such that $d_2 = h(d_1)$, we have that $ans(Qv_1, db(s_1)) \equiv ans(Qv_2, db(s_2))$.

Negation By induction hypothesis, for every (open) formula Φ and every valuations v_1 and v_2 that assign to each of its free variables a value $d_1 \in \text{ADOM}(db_1(s_1))$ and $d_2 \in \text{ADOM}(db_2(s_2))$, such that $d_2 = h(d_1)$, we have that $\Upsilon_1, s_1 \models \Phi v_1$ if and only if $\Upsilon_2, s_2 \models \Phi v_2$. By definition, $\Upsilon_1, s_1 \models (\neg\Phi) v_1$ if and only if $\Upsilon_1, s_1 \not\models \Phi v_1$, and, by induction hypothesis, $\Upsilon_1, s_1 \not\models \Phi v_1$ if and only if $\Upsilon_2, s_2 \not\models \Phi v_2$, which corresponds to $\Upsilon_2, s_2 \models (\neg\Phi) v_2$.

Conjunction By induction hypothesis, for every (open) formula Φ and every valuations v_1 and v_2 that assign to each of its free variables a value $d_1 \in \text{ADOM}(db_1(s_1))$ and $d_2 \in \text{ADOM}(db_2(s_2))$, such that $d_2 = h(d_1)$, we have that $\Upsilon_1, s_1 \models \Phi_i v_1$ if and only if $\Upsilon_2, s_2 \models \Phi_i v_2$, with $i \in \{1, 2\}$. Hence, $\Upsilon_1, s_1 \models \Phi_1 v_1$ and $\Upsilon_1, s_1 \models \Phi_2 v_1$ if and only if $\Upsilon_2, s_2 \models \Phi_1 v_2$ and $\Upsilon_2, s_2 \models \Phi_2 v_2$. By definition, we therefore have $\Upsilon_1, s_1 \models (\Phi_1 \wedge \Phi_2) v_1$ if and only if $\Upsilon_2, s_2 \models (\Phi_1 \wedge \Phi_2) v_2$.

Modal operator Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \approx_h s_2$. By definition, given a valuation v_1 that assigns to each free variable of Φ a value $d_1 \in \text{ADOM}(db_1(s_1))$, we have that $\Upsilon_1, s_1 \models (\langle - \rangle \Phi) v_1$ if there exists a transition $s_1 \Rightarrow_1 s_1'$ such that $\Upsilon_1, s_1' \models \Phi v_1$. Since $s_1 \approx_h s_2$, there exists a transition $s_2 \Rightarrow_2 s_2'$ such that $s_1' \approx_{h'} s_2'$, where h' extends h. By induction hypothesis, for every valuation v_2 that assigns to each free variable x of Φ a value $d_2 \in \text{ADOM}(db_2(s_2'))$, such that $d_2 = h'(d_1)$ with $x/d_1 \in v_1$, we have that $\Upsilon_1, s_1' \models \Phi v_1$ if and only if $\Upsilon_2, s_2' \models \Phi v_2$. Since h' is an extension of h, and v_1 assigns to each free variable of Φ a value $d_1 \in \text{ADOM}(db_1(s_1)) \subseteq \text{DOM}(h)$, we observe that for every pair of assignments $x/d_1 \in v_1$ and $x/d_2 \in v_2$,

it holds that $d_2 = h'(d_1) = h(d_1)$. Furthermore, since h induces an isomorphism between $db_1(s_1)$ and $db_2(s_2)$, for each assignment $x/d_2 \in v_2$, we have that $d_2 \in \textsc{adom}(db_2(s_2))$. Considering that $s_2 \Rightarrow_2 s_2'$, by definition we therefore get $\Upsilon_2, s_2 \models (\langle - \rangle \Phi) v_2$.

The other direction can be proven in a symmetric way.

Quantification Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \approx_h s_2$. By definition, given a formula Φ and a valuation v_1' that assigns to each free variable of Φ a value $d_1 \in \textsc{dom}(h)$, we have that $\Upsilon_1, s_1 \models (\exists x.\textsc{live}(x) \wedge \Phi) v_1'$ if and only if there exists $d \in \textsc{adom}(db_1(s_1))$ such that $\Upsilon_1, s_1 \models \Phi v_1$, where $v_1 = v_1'[x/d]$. By induction hypothesis, for every valuation v_2 that assigns to each free variable y of Φ a value $d_2 \in \textsc{adom}(db_2(s_2))$, such that $d_2 = h(d_1)$ with $y/d_1 \in v_1$, we have that $\Upsilon_1, s_1 \models \Phi v_1$ if and only if $\Upsilon_2, s_2 \models \Phi v_2$. More specifically, the structure of v_2 is $v_2 = v_2'[x/d']$, where $d' = h(d) \in \textsc{adom}(db_2(s_2))$ because h induces an isomorphism between $db_1(s_1)$ and $db_2(s_2)$. Hence, we get $\Upsilon_2, s_2 \models (\exists x.\textsc{live}(x) \wedge \Phi) v_2'$.

The other direction can be proven in a symmetric way. □

PROOF OF THEOREM 3.1. We prove the theorem in two steps. First, we show that Lemma A.1 can be extended to the infinitary version of $\mathcal{L}_A$ that supports arbitrary countable disjunction. Then, we recall that fixpoints can be translated into this infinitary logic, thus guaranteeing invariance for the whole $\mu\mathcal{L}_A$ logic.

Let Ψ be a countable ordered set of open $\mathcal{L}_A$ formulae. Given a transition system $\Upsilon = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$, the semantics of $\bigvee \Psi$ is $(\bigvee \Psi)^\Upsilon_v = \bigcup_{\psi \in \Psi} (\psi)^\Upsilon_v$. Therefore, given a state s of Υ and a variable valuation v that assigns to each free variable of Ψ a value $d \in \textsc{adom}(db(s))$, we have $\Upsilon, s \models \Psi v$ if and only if $\Upsilon, s \models \psi v$ for some $\psi \in \Psi$. Arbitrary countable conjunction is obtained for free because of negation.

We show that the invariance result proven in Lemma A.1 trivially extends to this arbitrary countable disjunction. Lemma A.1 guarantees that invariance is preserved for any finite disjunction. Formally, let $\{\Phi_1, \ldots, \Phi_n\}$ be a finite set of open $\mathcal{L}_A$ formulae. Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \approx_h s_2$. Then, for every valuations v_1 and v_2 that assign to each free variable of $\{\Phi_1, \ldots, \Phi_n\}$ a value $d_1 \in \textsc{adom}(db_1(s_1))$ and $d_2 \in \textsc{adom}(db_2(s_2))$, such that $d_2 = h(d_1)$, we have that $\Upsilon_1, s_1 \models (\bigvee_{i \in \{1,\ldots,n\}} \Phi_i) v_1$ if and only if $\Upsilon_2, s_2 \models (\bigvee_{i \in \{1,\ldots,n\}} \Phi_i) v_2$.

Now consider two valuations v_1' and v_2' that assign to each free variable of $\bigvee \Psi$ a value $d_1 \in \textsc{adom}(db_1(s_1))$ and $d_2 \in \textsc{adom}(db_1(s_2))$, such that $d_2 = h(d_1)$. By definition, $\Upsilon_1, s_1 \models (\bigvee \Psi) v_1'$ if and only if there exists $\psi_k \in \Psi$ such that $\Upsilon_1, s_1 \models \psi_k v_1'$. The proof of invariance for the infinitary $\mathcal{L}_A$ logic is then obtained by observing that $\Upsilon_1, s_1 \models (\bigvee \Psi) v_1'$ if and only if $\Upsilon_1, s_1 \models (\bigvee_{i \in \{1,\ldots,k\}} \psi_i) v_1'$ if and only if $\Upsilon_2, s_2 \models (\bigvee_{i \in \{1,\ldots,k\}} \psi_i) v_2'$ if and only if $\Upsilon_2, s_2 \models (\bigvee \Psi) v_2'$.

In order to extend the result to the whole $\mu\mathcal{L}_A$, we resort to the well-known result stating that fixpoints of the μ-calculus can be translated into the infinitary Hennessy Milner logic by iterating over *approximants*, where the approximant of index α is denoted by $\mu^\alpha Z.\Phi$ ($\nu^\alpha Z.\Phi$). This is a standard result that also holds for $\mu\mathcal{L}_A$. In particular, approximants are built as follows:

$$\mu^0 Z.\Phi = \mathsf{false} \qquad \nu^0 Z.\Phi = \mathsf{true}$$
$$\mu^{\beta+1} Z.\Phi = \Phi[Z/\mu^\beta Z.\Phi] \qquad \nu^{\beta+1} Z.\Phi = \Phi[Z/\nu^\beta Z.\Phi]$$
$$\mu^\lambda Z.\Phi = \bigvee_{\beta<\lambda} \mu^\beta Z.\Phi \qquad \nu^\lambda Z.\Phi = \bigwedge_{\beta<\lambda} \nu^\beta Z.\Phi$$

where λ is a limit ordinal, and where fixpoints and their approximants are connected by the following properties: given a transition system Υ and a state s of Υ

- $s \in (\mu Z.\Phi)^\Upsilon_{v,V}$ if and only if there exists an ordinal α such that $s \in (\mu^\alpha Z.\Phi)^\Upsilon_{v,V}$ and, for every $\beta < \alpha$, it holds that $s \notin (\mu^\beta Z.\Phi)^\Upsilon_{v,V}$;
- $s \notin (\nu Z.\Phi)^\Upsilon_{v,V}$ if and only if there exists an ordinal α such that $s \notin (\nu^\alpha Z.\Phi)^\Upsilon_{v,V}$ and, for every $\beta < \alpha$, it holds that $s \in (\nu^\beta Z.\Phi)^\Upsilon_{v,V}$. □

A.2 Persistence Preserving Mu-Calculus

We prove *persistence preserving bisimulation invariance* for $\mu\mathcal{L}_P$. To prove the invariance result, we adopt a two-step approach. We first prove the result for the logic $\mathcal{L}_P$, obtained from $\mu\mathcal{L}_P$ dropping the predicate variables and the fixpoint constructs. Such a logic corresponds to a first-order variant of the Hennessy Milner logic; note that the semantics of this logic is completely independent from the second-order valuation. We then extend the result to the whole $\mu\mathcal{L}_P$ by dealing with fixpoints.

LEMMA A.2. *Consider two transition systems* $\Upsilon_1 = \langle \Delta_1, \mathcal{R}, \Sigma_1, s_{01}, db_1, \Rightarrow_1 \rangle$ *and* $\Upsilon_2 = \langle \Delta_2, \mathcal{R}, \Sigma_2, s_{02}, db_2, \Rightarrow_2 \rangle$, *a partial bijection* h *between* Δ_1 *and* Δ_2 *and two states* $s_1 \in \Sigma_1$ *and* $s_2 \in \Sigma_2$ *such that* $s_1 \sim_h s_2$. *Then for every (open) formula* Φ *of* $\mathcal{L}_P$, *and every valuations* v_1 *and* v_2 *that assign to each of its free variables a value* $d_1 \in \textsc{adom}(db_1(s_1))$ *and* $d_2 \in \textsc{adom}(db_2(s_2))$, *such that* $d_2 = h(d_1)$, *we have that*

$$\Upsilon_1, s_1 \models \Phi v_1 \textit{ if and only if } \Upsilon_2, s_2 \models \Phi v_2.$$

PROOF. We proceed by induction on the structure of Φ. In particular, we discuss the two base cases of $\langle - \rangle(\textsc{live}(x) \wedge \Phi)$ and $[-](\textsc{live}(x) \wedge \Phi')$ with one variable. For convenience, we rewrite the latter case to $\langle - \rangle(\textsc{live}(x) \rightarrow \Phi)$, where $\Phi = \neg\Phi'$. The other cases are derived, or proven in the same way as done for Lemma A.1.

Modal operator (conjunction) Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_h s_2$. Let x be the only free variable of Φ, and x/d a valuation such that $d \in \textsc{adom}(db_1(s_1))$. Then, by definition we have that $\Upsilon_1, s_1 \models (\langle - \rangle(\textsc{live}(x) \wedge \Phi)[x/d]$ if there exists a transition $s_1 \Rightarrow_1 s_1'$ such that $d \in \textsc{adom}(db_1(s_1'))$ and $\Upsilon_1, s_1' \models \Phi[x/d]$. Since $s_1 \sim_h s_2$, there exists a transition $s_2 \Rightarrow_2 s_2'$ such that $s_1' \sim_{h'} s_2'$, where h' is compatible with h. By induction hypothesis and by considering that h' is an isomorpshim between $db_1(s_1')$ and $db_2(s_2')$, we have that $\Upsilon_1, s_1' \models \Phi[x/d]$ if and only if $h'(d) \in \textsc{adom}(db_2(s_2'))$ and $\Upsilon_2, s_2' \models \Phi[x/h'(d)]$. Now we observe that $d \in \textsc{adom}(db_1(s_1)) \cap \textsc{adom}(db_1(s_1'))$ and h' is an extension of $h|_{\textsc{adom}(db_1(s_1)) \cap \textsc{adom}(db_1(s_1'))}$. This implies that $h'(d) = h(d) \in \textsc{adom}(db_2(s_2))$, because h is an isomorphism between $db_1(s_1)$ and $db_2(s_2)$. Considering that $s_2 \Rightarrow_2 s_2'$, by definition we therefore get $\Upsilon_2, s_2 \models (\langle - \rangle(\textsc{live}(x) \wedge \Phi))[x/h(d)]$.

The other direction can be proven in a symmetric way.

Modal operator (implication) Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_h s_2$. Let x be the only free variable of Φ, and x/d a valuation such that $d \in \textsc{adom}(db_1(s_1))$. Then, by definition we have that $\Upsilon_1, s_1 \models (\langle - \rangle(\textsc{live}(x) \rightarrow \Phi)[x/d]$ if there exists a transition $s_1 \Rightarrow_1 s_1'$ such that $d \notin \textsc{adom}(db_1(s_1'))$ or $\Upsilon_1, s_1' \models \Phi[x/d]$. Since $s_1 \sim_h s_2$, there exists a transition $s_2 \Rightarrow_2 s_2'$ such that $s_1' \sim_{h'} s_2'$, where h' is an extension of $h|_{\textsc{adom}(db_1(s_1)) \cap \textsc{adom}(db_1(s_1'))}$ Now we

discuss the two cases in which $d \notin \text{ADOM}(db_1(s'_1))$ and $d \in \text{ADOM}(db_1(s'_1))$.

- Assume that $d \notin \text{ADOM}(db_1(s'_1))$. Since $s_1 \sim_h s_2$, we have that $h(d) \in \text{ADOM}(db_2(s_2))$. Now, towards contradiction, let us assume that $h(d) \in \text{ADOM}(db_2(s'_2))$. Hence, we have $h(d) \in \text{ADOM}(db_2(s_2)) \cap \text{ADOM}(db_2(s'_2))$. Observe that h' is an extension of $h|_{\text{ADOM}(db_1(s_1)) \cap \text{ADOM}(db_1(s'_1))}$, which is equivalent to state that h'^{-1} is an extension of $h^{-1}|_{\text{ADOM}(db_2(s_2)) \cap \text{ADOM}(db_2(s'_2))}$. This implies that $h^{-1}(d) = h'^{-1}(d) = d$. Since h' is an isomorphism between $db_1(s'_1)$ and $db_2(s'_2)$, then $d \in \text{ADOM}(db_1(s'_1))$, and this contradicts the hypothesis.
- Assume that $d \in \text{ADOM}(db_1(s'_1))$. Then we can proceed following the line of reasoning used for the case of $\langle - \rangle(\text{LIVE}(x) \wedge \Phi)$.

The other direction can be proven in a symmetric way. □

PROOF OF THEOREM 3.2. The proof is analogous to that of Theorem 3.1, but now using Lemma A.2. □

B. DETERMINISTIC SERVICES

B.2 Run-Bounded Systems

PROOF OF THEOREM 4.1. The proof is by reduction from the halting problem. Given a deterministic Turing Machine *TM*, we define DCDS $\mathcal{S}$ with deterministic services and propositional safety property Φ, such that *TM* halts if and only if $\Upsilon_{\mathcal{S}} \models \Phi$.

Intuitively, every run of $\Upsilon_{\mathcal{S}}$ simulates a run of *TM*. Each state s of $\Upsilon_{\mathcal{S}}$ models a configuration of *TM*. A transition in $\Upsilon_{\mathcal{S}}$ models a transition in *TM*. We give the construction next.

The DCDS. To model a configuration of *TM* in a relation of the DCDS state, we model the visited tape segment as a graph whose nodes are cell identifiers, and whose edges form a linear path. The edge relation is called $right$, with the intended meaning that $right(x, y)$ declares cell y to be the right neighbor of cell x on the tape. We also introduce a relation sym, with $sym(c, s)$ intended to model that cell c holds symbol s. Unary relation $head$ models the head position: $head(c)$ means that the head points to cell c. Finally, unary relation $state$ keeps the state of *TM*, and a boolean predicate $halted$ is meant to detect that *TM* has halted. In summary, the data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ of $\mathcal{S}$ contains the schema $\mathcal{R} = \{right/2, sym/2, head/1, halted/0\}$. We detail $\mathcal{E}$ and $\mathcal{I}_0$ after sketching the process layer.

There is a single action α, in charge of simulating the transitions of *TM*. It has no parameters, and its guard is always true: $\texttt{true} \mapsto \alpha$. α contains the following effects.

e_{copy} simply copies the part of the tape that stays unchanged in the transition because the head doesn't point to it:

$$
\begin{aligned}
e_{copy} : \; & right(X,Y) \wedge right(Y,Z) \wedge \\
& sym(X,SX) \wedge sym(Y,SY) \wedge sym(Z,SZ) \wedge \\
& \neg(head(X) \wedge head(Y) \wedge head(Z)) \\
& \rightsquigarrow \\
& \{right(X,Y), right(Y,Z), \\
& \; sym(X,SX), sym(Y,SY), sym(Z,SZ)\}
\end{aligned}
$$

In addition, we add effects for each entry of *TM*'s transition relation δ.

For instance, if $(p, b, \rightarrow) \in \delta(s, a)$ (i.e. δ prescribes that in state s, if the head points to a cell containing symbol a, *TM* changes state to p, the cell's symbol is overwritten with b, and the head moves to the right), we introduce two effects. One for the case when the tape needs no extension to the right,

$$
\begin{aligned}
e^{noext}_{s,a,p,b,\rightarrow} : \; & right(X,Y) \wedge sym(X,a) \wedge sym(Y,SY) \wedge SY \neq \omega \wedge \\
& head(X) \wedge state(s) \\
& \rightsquigarrow \\
& \{right(X,Y), sym(X,b), sym(Y,SY), \\
& \; head(Y), state(p)\},
\end{aligned}
$$

and one when it does:

$$
\begin{aligned}
e^{ext}_{s,a,p,b,\rightarrow} : \; & right(X,Y) \wedge sym(X,a) \wedge sym(Y,\omega) \wedge \\
& head(X) \wedge state(s) \\
& \rightsquigarrow \\
& \{right(X,Y), right(Y, newCell(Y)), \\
& \; sym(X,b), sym(Y,\perp), sym(newCell(Y),\omega), \\
& \; head(Y), state(p)\}.
\end{aligned}
$$

To distinguish among constants and variables in the above effect specifications, we use capital letters for the latter and lower-case letters for the former. Notice that the extension is performed by calling service $newCell$, which is meant to return a fresh cell id (we show below how to ensure this). Also notice the use of special symbol ω, which is reserved for labeling the end of the tape segment. Finally, special symbol $\perp$ is by convention used to initialize the tape prior to starting the run.

We are not quite done, as we still need to ensure that $right$ induces a linear order on the collection of cell identifiers generated during the run. Notice that this cannot be achieved exclusively by declaring FO constraints in $right$, as linear orders are not FO-axiomatizable. The solution must exploit the interplay between constraints on $right$ and the way $\Upsilon_{\mathcal{S}}$ transitions.

Observe that, by definition of the effects that extend $right$ (e.g. $e^{ext}_{s,a,p,b,\rightarrow}$ above), at each step the current right end of the tape segment obtains at most one new successor. However, if the call of service $newCell$ returns a cell id that already appears in the tape segment, then there can be some cell with several predecessors according to $right$. We rule out this case by declaring the second component of $right$ to be a key. It follows that $right$ must be either (i) a linear path (possibly starting from a source node that has a self-loop), or (ii) it must contain a simple cycle involving more than one cell id. The simple cycle is created at the step when $newCell$ returns the id of the leftmost cell.

We wish to force case (i). To rule out case (ii), we proceed as follows: we initialize $right$ to contain a source node 0 that can never be a cell id because it cannot be returned by $newCell$ without violating the key constraint on $right$. To this end, we initialize $\mathcal{I}_0$ to

- $right^{\mathcal{I}_0} = \{(0,0), (0,1), (1,2)\}$,
- $sym^{\mathcal{I}_0} = \{(1,\$), (2,\omega)\}$,
- $head^{\mathcal{I}_0} = \{2\}$,
- $state^{\mathcal{I}_0} = \{s_0\}$,
- $halted^{\mathcal{I}_0} = \{\}$,

where s_0 is the initial state of *TM*.

Notice that, if we disregard cell 0, $\mathcal{I}_0$ contains the representation of an empty tape (symbol \$ labels the left end, symbol ω the right end). Also notice that $right^{\mathcal{I}_0}$ has type (i). An easy induction shows that every run prefix must also construct a $right$ relation of type (i), since any attempt to extend $right$ with an edge back to one of its existing nodes violates the key constraint.

Because symbol \$ denotes the left end of the tape, it also follows easily from the behavior of *TM* that during the run, the head will never reach the special cell 0, so $head$ can only take values from the suffix of $right$ starting at cell 1, which is a true linear path.

Now assume without loss of generality that the *TM* is normalized to enter a particular sink state h when it halts. We add effect

e_h, which detects the halting state and sets the boolean predicate $halted$.

Observe that for the cases when the head stays in place or moves left, no tape extension is required, so each such entry in the transition relation corresponds to a single effect.

The property. We define the propositional safety property Φ as

$$\Phi : \mathbf{G}\neg halted.$$

It is easy to see that the runs of $\Upsilon_{\mathcal{S}}$ correspond ono-to-one to the runs of *TM*. Since Φ is a linear-time property, this run correspondence suffices to guarantee that $\Upsilon_{\mathcal{S}} \models \Phi$ if and only if *TM* does not halt. □

PROOF OF THEOREM 4.2. The proof is directly obtained from Theorem 4.4, noticing that model checking of propositional μ-calculus formulae over finite transition systems is decidable [22]. □

Proof of Theorem 4.3. In view of proving this result, we first introduce a key lemma. We say that a transition system is *adom-inflationary* if the active domain of every state is included in its successor's active domain. We say that a DCDS $\mathcal{S}$ is adom-inflationary if $\Upsilon_{\mathcal{S}}$ is adom-inflationary. We can show that, for adom-inflationary transition systems, persistence-preserving bisimilarity coincides with history-preserving bisimilarity.

LEMMA B.1. *Consider two adom-inflationary DCDSs with non-deterministic services, $\mathcal{S}_1, \mathcal{S}_2$. Then $\Upsilon_{\mathcal{S}_1} \sim \Upsilon_{\mathcal{S}_2}$ if and only if $\Upsilon_{\mathcal{S}_1} \approx \Upsilon_{\mathcal{S}_2}$.*

PROOF OF LEMMA B.1. A comparison the two notions of bisimilarity reveals that the difference is in the local condition, as follows.

Notice first that what both bisimilarity notions have in common is that they mention bisimilar states s_1 and s_2 and witness isomorphism h, and their successors $s_1 \Longrightarrow s_1'$, $s_1 \Longrightarrow s_2'$ such that s_1' and s_2' are bisimilar as witnessed by isomorphism h'. The key difference lies in how h' and h are related. In the history-preserving flavor, h' must extend h, while in the persistence-preserving flavor h' must only extend $h\mid_{\textsc{adom}(s_1)\cap\textsc{adom}(s_1')}$.

Clearly, history-preserving bisimilarity implies persistence-preserving bisimilarity. However, notice that if the transition systems are adom-inflationary, then the converse also holds. Indeed, assume $s_1 \sim_h s_2$. By definition, h' extends $h\mid_{\textsc{adom}(s_1)\cap\textsc{adom}(s_1')}$. But because of adom-inflation, $\textsc{adom}(s_1) \subseteq \textsc{adom}(s_1')$ and hence $\textsc{adom}(s_1)\cap\textsc{adom}(s_1') = \textsc{adom}(s_1)$, yielding $h\mid_{\textsc{adom}(s_1)\cap\textsc{adom}(s_1')} = h$. Hence, h' extends h, which is the condition for history-preserving bisimilarity. □

PROOF OF THEOREM 4.3. We prove the result by exploiting the reduction postulated by Theorem 6.1.

Starting from run-bounded DCDS D with deterministic services, the reduction gives us state-bounded DCDS N with non-deterministic services. Moreover, the two transition systems have the same domains, and the projection of Υ_N on the schema of D coincides with Υ_D (Theorem 6.1(ii)). In more detail, denoting the schema of D with $\mathcal{R}_D$ and the schema of N with $\mathcal{R}_N$, there is a bijection β between the states of Υ_D and the states of Υ_N, such that $s = \beta(s)\mid_{\mathcal{R}_D}$. Clearly, this implies that Υ_D and Υ_N satisfy the same $\mu\mathcal{L}$ formulae.

However, a weaker statement suffices for our purpose. By definition of history-preserving bisimilarity, Theorem 6.1(ii) implies that

(1) $\Upsilon_D \approx \Upsilon_N$.

We recall that on the way to proving Theorem 5.3, it is shown in Theorem 5.4 that since N is state-bounded, we can construct using algorithm RCYCL a finite-state abstract transition system F such that $\Upsilon_N \sim \Upsilon_F$ (F is an eventually recycling pruning of Υ_N).

An inspection of the reduction in Theorem 6.1 reveals that Υ_N is adom-inflationary. But since $\Upsilon_N \sim \Upsilon_F$, it follows that Υ_F is adom-inflationary as well (by the local condition of persistence-preserving bisimilarity). Thus Lemma B.1 applies, yielding

(2) $\Upsilon_N \approx \Upsilon_F$.

By (1) and (2), and by transitivity of $\approx$, we obtain that $\Upsilon_D \approx \Upsilon_F$. □

PROOF OF THEOREM 4.4. Theorem 4.3 implies that, given a DCDS $\mathcal{S}$, there exists a finite-state transition system $\Theta_{\mathcal{S}} = \langle \mathbb{U}, \mathcal{R}, \Sigma_a, s_0^a, db_a, \Longrightarrow_a \rangle$ that is history preserving bisimilar to the concrete transition system $\Upsilon_{\mathcal{S}} = \langle \mathbb{U}, \mathcal{R}, \Sigma, s_0, db, \Longrightarrow \rangle$. Thus, it is possible to use $\Theta_{\mathcal{S}}$ in place of $\Upsilon_{\mathcal{S}}$ for verification. In particular, given a $\mu\mathcal{L}_A$ property Φ, the verification problem is reduced to $\Theta_{\mathcal{S}} \models \Phi$. Let $\textsc{adom}(\Theta_{\mathcal{S}}) = \bigcup_{s_i\in\Sigma} \textsc{adom}(db(s_i))$. If $\Theta_{\mathcal{S}}$ is finite-state, then there exists a bound b such that $|\textsc{adom}(\Theta_{\mathcal{S}})| < b$. Consequently, it is possible to transform Φ into an equivalent *finite* propositional μ-calculus formula $\textsc{prop}(\Phi)$ as follows:

$$\begin{aligned}
\textsc{prop}(Q) &= Q \\
\textsc{prop}(\neg\Psi) &= \neg\textsc{prop}(\Psi) \\
\textsc{prop}(\Psi_1 \wedge \Psi_2) &= \textsc{prop}(\Psi_1) \wedge \textsc{prop}(\Psi_2) \\
\textsc{prop}(\langle - \rangle \Psi) &= \langle - \rangle \textsc{prop}(\Psi) \\
\textsc{prop}(Z) &= Z \\
\textsc{prop}(\mu Z.\Psi) &= \mu Z.\textsc{prop}(\Psi) \\
\textsc{prop}(\exists x.\textsc{live}(x) \wedge \Psi(x)) &= \bigvee_{t_i \in \textsc{adom}(\mathcal{S})} \textsc{live}(t_i) \wedge \textsc{prop}(\Psi(t_i))
\end{aligned}$$

Clearly, $\Theta_{\mathcal{S}} \models \Phi$ if and only if $\Theta_{\mathcal{S}} \models \textsc{prop}(\Phi)$. The proof is then obtained by observing that verification of μ-calculus formulae over finite transition systems is decidable [22]. □

PROOF OF THEOREM 4.5. The Theorem is proved by exhibiting, for every n, a $\mu\mathcal{L}$ property that requires the existence of at least n objects in the transition system.

Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \emptyset, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, where $\mathcal{F} = \{f/1\}$, $\mathcal{R} = \{R/1, Q/1\}$, $\mathcal{I}_0 = \{R(a)\}$, $\varrho = \{R(x) \mapsto \alpha(x)\}$ and $\mathcal{A} = \{\alpha(p)\}$, where $\alpha(p) : \{\mathsf{true} \rightsquigarrow \{Q(f(p))\}\}$. The concrete transition system $\Upsilon_{\mathcal{S}}$ has the following shape:

- The initial state is $s_0 = \langle \{R(a)\}, \emptyset \rangle$;
- s_0 is connected to infinitely many successor states, each one storing into Q a distinct value d resulting from the service call $f(a)$; each such state has then the form $s_d = \langle \{Q(d)\}, \{f(a) \mapsto d\} \rangle$;
- each s_d has no outgoing edge, because there is no applicable action in s_d.

$\mathcal{S}$ is clearly run-bounded, in particular by a bound $b = 3$.

Let us now consider the following $\mu\mathcal{L}$ property without fixpoints:

$$\Phi_n = \exists x_1, \ldots, x_n. \bigwedge_{i\neq j} x_i \neq x_j \wedge \bigwedge_{i\in\{1,\ldots,n\}} \langle - \rangle Q(x_i)$$

The property states that there are n distinct values, each of which is stored into relation Q in one of the successors of the initial state. It is easy to see that $\Upsilon_{\mathcal{S}} \models \Phi_n$ for every n. On the other hand, for every finite state abstraction $\Theta_{\mathcal{S}}$ with k successors of the initial state, we have that $\Theta_{\mathcal{S}} \not\models \Phi_{k+1}$. □

B.3 Weakly Acyclic DCDSs

PROOF OF THEOREM 4.6. The proof is by reduction from the halting problem. We reuse without change the reduction in the proof of Theorem 4.1. This reduction yields for any Turing Machine *TM* a DCDS with deterministic services $\mathcal{S}$, such that $\mathcal{S}$ simulates *TM*'s computation. That is, the runs of *TM* correspond one-to-one to the runs of $\Upsilon_{\mathcal{S}}$. It follows immediately that *TM* halts if and only if $\mathcal{S}$ is run-bounded. □

PROOF OF LEMMA 4.1. Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$. Consider now $\Upsilon_{\mathcal{S}} = \langle \mathcal{C}, \mathcal{R}, \Sigma, s_0, db, \Longrightarrow \rangle$ and $\Upsilon_{\mathcal{S}^+} = \langle \mathcal{C}, \mathcal{R}, \Sigma^+, s_0, db, \Longrightarrow^+ \rangle$. Since $\mathcal{S}^+$ is weakly acyclic by hypothesis, to prove that run boundedness of $\Upsilon_{\mathcal{S}^+}$ implies run boundedness of $\Upsilon_{\mathcal{S}}$, we show the following stronger result: for every run τ in $\Upsilon_{\mathcal{S}}$, there exists a run τ^+ in $\Upsilon_{\mathcal{S}^+}$ such that, for all pairs of states $\tau(i) = \langle \mathcal{I}_i, \mathcal{M}_i \rangle$ and $\tau^+(i) = \langle \mathcal{I}_i^+, \mathcal{M}_i^+ \rangle$, we have

1. $\mathcal{M}_i^+$ extends $\mathcal{M}_i$;
2. $\mathcal{I}_i \subseteq \mathcal{I}_i^+$;
3. for the mappings mentioned in $\mathcal{M}_i^+$ but not in $\mathcal{M}_i$, $\mathcal{M}_i^+$ "agrees" with the maps contained in the suffix of $\tau[i]$, i.e.,

$$\mathcal{M}_i^+|_{C_i} = (\bigcup_{j>i} \mathcal{M}_j)|_{C_i}$$

where $C_i = \text{DOM}(\mathcal{M}_i^+) \cap \bigcup_{j>i} \text{DOM}(\mathcal{M}_j)$.

We prove this by induction on the length of τ:

(base case) The initial state of both runs is $\tau(0) = \tau^+(0) = \langle \mathcal{I}_0, \emptyset \rangle$, and therefore all the three conditions are trivially satisfied.

(inductive step) Consider a pair of corresponding states $\tau(i)$ and $\tau^+(i)$, with $i > 0$. By definition, $\tau(i) \Longrightarrow \tau(i+1)$ means that there exists an action $\alpha \in \mathcal{A}$ and a substitution σ for the parameters of α such that $\langle \tau(i), \alpha\sigma, \tau(i+1) \rangle \in \text{EXEC}_{\mathcal{S}}$. We first observe that α^+ can be executed in $\tau^+(i)$, since $\mathcal{P}^+$ does not impose any restriction on the executability of actions. Let $Next^+ = \{s^+ \in \Sigma^+ \mid \langle \tau^+(i), \alpha, s^+ \rangle \in \text{EXEC}_{\mathcal{S}^+}\}$ be the set of successor states of $\tau^+(i)$ that are obtained from the application of α^+.

We now show that there exists $\underline{s} \in Next^+$ that satisfies the three claims above. The proof is then obtained by simply imposing $\tau^+(i+1) = \underline{s}$.

1. By definition, $\text{DOM}(\mathcal{M}_{i+1}) = \text{DOM}(\mathcal{M}_i) \cup \text{CALLS}(\text{DO}(\mathcal{I}_i, \alpha\sigma))$, and, for every $s_k = \langle \mathcal{M}_k^+, \mathcal{I}_k^+ \rangle \in Next^+$, we have $\text{DOM}(\mathcal{M}_k^+) = \text{DOM}(\mathcal{M}_i^+) \cup \text{CALLS}(\text{DO}(\mathcal{I}_i^+, \alpha^+\sigma))$. Consider each effect specification $q_j^+ \wedge Q_j \rightsquigarrow E_j \in \text{EFFECT}(\alpha)$. By definition of q_j^+ and Q_j^-, $\theta \in ans((q_j^+ \wedge Q_j^+)\sigma, \mathcal{I}_i)$ implies $\theta \in ans(q_j^+\sigma, \mathcal{I}_i)$, which in turn implies $\theta \in ans(q_j^+\sigma, \mathcal{I}_i^+)$, because $\mathcal{I}_i \subseteq \mathcal{I}_i^+$ by induction hypothesis. Consequently, we have $\text{DO}(\mathcal{I}_i, \alpha\sigma) \subseteq \text{DO}(\mathcal{I}_i^+, \alpha^+\sigma)$, and hence $\text{CALLS}(\text{DO}(\mathcal{I}_i, \alpha\sigma)) \subseteq \text{CALLS}(\text{DO}(\mathcal{I}_i^+, \alpha^+\sigma))$. Since $\text{DOM}(\mathcal{M}_i) \subseteq \text{DOM}(\mathcal{M}_i^+)$ by induction hypothesis, then we obtain $\text{DOM}(\mathcal{M}_{i+1}) \subseteq \text{DOM}(\mathcal{M}_k^+)$. Since $\mathcal{S}^+$ has no equality constraint, the states in $Next^+$ cover every possible result obtained by calling the service call in $\mathcal{M}_k^+ \setminus \mathcal{M}_i^+$, including those states for which $\mathcal{M}_k^+$ is an extension of $\mathcal{M}_{i+1}$. We use $\underline{Next}^+$ to denote such states.
2. By definition, for each state $s_k = \langle \mathcal{M}_k^+, \mathcal{I}_k^+ \rangle \in \underline{Next}^+$, we have that $\mathcal{M}_k^+$ extends $\mathcal{M}_{i+1}$. Therefore, since $\text{DO}(\mathcal{I}_i, \alpha\sigma) \subseteq \text{DO}(\mathcal{I}_i^+, \alpha^+\sigma)$, we have $\mathcal{I}_{i+1} = \mathcal{M}_{i+1}(\text{DO}(\mathcal{I}_i, \alpha\sigma)) \subseteq \mathcal{I}_k^+ = \mathcal{M}_k^+(\text{DO}(\mathcal{I}_i^+, \alpha^+\sigma))$.
3. Since $\mathcal{S}^+$ has no equality constraints, we observe that the states in $\underline{Next}^+$ cover all possible values for the service calls that are not mentioned in $\mathcal{M}_{i+1}$. Therefore, there must exist at least one state $\underline{s} \in \underline{Next}^+$ that satisfies the third claim. In other words, by imposing $\tau^+(i+1) = \underline{s}$, we have

$$\mathcal{M}_{i+1}^+|_{C_{i+1}} = (\bigcup_{j>i+1} \mathcal{M}_j)|_{C_{i+1}} \quad \square$$

PROOF OF THEOREM 4.7. Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$. We consider the positive approximate $\mathcal{S}^+$, showing that if the the dependency graph $G = \langle N, E \rangle$ of $\mathcal{S}$ (which corresponds by definition to the one of $\mathcal{S}^+$) is weakly acyclic, then $\mathcal{S}^+$ is run-bounded. The complete proof is then directly obtain by appealing to Lemma 4.1, which states that if $\mathcal{S}^+$ is run-bounded, then $\mathcal{S}^+$ is run-bounded as well.

To prove that weak acyclicity of $\mathcal{S}$ implies that $\mathcal{S}^+$ is run-bounded, we exploit the connection with the chase of a set of tuple generating dependencies (TGDs) in data exchange. In particular, we resort to the proof given in [23], Theorem 3.9. For every node $p \in N$, we consider an incoming path to be any (finite or infinite) path ending in p. For simplicity, we say that a value appears in position $p = \langle R_k, j \rangle \in N$ if it appears in the j-th component of an R_k tuple. We define the rank of p, denoted $\underline{\text{rank}}(p)$, as the maximum number of special edges on any such incoming path. Since $\mathcal{S}^+$ is weakly acyclic by hypothesis, G does not contain cycles going through special edges, and therefore $\underline{\text{rank}}(p)$ is finite. Let r be the maximum among $\underline{\text{rank}}(p_i)$ over all nodes. We observe that $r \leq |N|$; indeed no path can lead to the same node twice using special edges, otherwise G would contain a cycle going through special edges, thus breaking the weak acyclicity hypothesis. Notice also that $|N|$ is a constant value, because it is obtained from $\mathcal{R}$, which is fixed. We now partition the nodes in N according to their rank, obtaining a set of sets $\{N_0, N_1, \ldots, N_r\}$, where N_i is the set of all nodes with rank i. The proof is then a natural consequence of the following claim:

> **Claim.** Consider a trace τ in $\Upsilon_{\mathcal{S}^+}$. For every $i \in \{1, \ldots, r\}$, the total number of distinct values occurring in the databases of τ inside position $p \in N_i$ is bounded by a polynomial $P_i(|\text{ADOM}(\mathcal{I}_0)|)$.

We prove the claim by induction on i:

(Base case) Consider $p \in N_0$. By definition, p has no incoming path containing special edges. Therefore, no new values are stored in p along the run: p can just store values that are part of the initial database $\mathcal{I}_0$. This holds for all nodes in N_0, and hence we can fix $P_0(|\text{ADOM}(\mathcal{I}_0)|) = |\text{ADOM}(\mathcal{I}_0)|$.

(Inductive step) Consider $p \in N_i$, with $i \in \{1, \ldots, r\}$. The first kind of values that may be stored inside p are those values that were stored inside the component itself in $\mathcal{I}_0$. The number of such values is at most $|\text{ADOM}(\mathcal{I}_0)|$. In addition, a value may be stored in p for two reasons: either it is copied from some other position $p' \in N_j$ with $i \neq j$, or it is generated by means of a service call.

We first determine how many fresh values can be generated by service calls. The possibility of generating and storing a new value in p as a result of an action is reflected by

the presence of special edges. By definition, any special edge entering p must start from a node $p' \in N_0 \cup \ldots \cup N_{i-1}$. By induction hypothesis, the number of distinct values that can exist in p' is bounded by $H(|\text{ADOM}(\mathcal{I}_0)|) = \sum_{j\in\{0,\ldots,i-1\}} P_j(|\text{ADOM}(\mathcal{I}_0)|)$. Let b_a be the maximum number of special edges that enter a position, over all positions in the schema; b_a bounds the arity taken by service calls in $\mathcal{F}$. Then for every choice of b_a values in $N_0 \cup \ldots \cup N_{i-1}$ (one for each special edge that can enter a position) and for every action in $\mathcal{A}^+$, the number of new values generated at position p is bounded by $t_f \cdot H(n)^{b_a}$, where t_f is the total number of facts mentioned in the effects of actions that belong to $\mathcal{A}^+$. Notice that this number does not depend on the data in $\mathcal{I}_0$. By considering all positions in N_i, the total number of values that can be generated is then bounded by $G(|\text{ADOM}(\mathcal{I}_0)|) = |N_i| \cdot t_f \cdot H(|\text{ADOM}(\mathcal{I}_0)|)^{b_a}$. Obviously, $G(\cdot)$ is a polynomial, because t_f and b_a are values extracted from the schema $\mathcal{R}$ of the DCDS, which is fixed.

We count next the number of distinct values that can be copied to positions of N_i from positions of N_j, with $j \neq i$. A copy is represented in the graph as a normal edge going from a node in N_j to a node in N_i, with $j \neq i$. We observe first that such normal edges can start only from nodes in $N_0 \cup \ldots \cup N_{i-1}$, that is, they cannot start from nodes in N_j with $j > i$. We prove this by contradiction. Assume that there exists $\langle p', p, \mathsf{false}\rangle \in E$, such that $p \in N_i$ and $p' \in N_j$ with $j > i$. In this case, the rank of p would be $j > i$, which contradicts the fact that $p \in N_i$. As a consequence, the number of distinct values that can be copied to positions in N_i is bounded by the total number of values in $N_0 \cup \ldots \cup N_{i-1}$, which corresponds to $H(|\text{ADOM}(\mathcal{I}_0)|)$ from our previous consideration. Putting it all together, we define $P_i(|\text{ADOM}(\mathcal{I}_0)|) = |\text{ADOM}(\mathcal{I}_0)| + G(|\text{ADOM}(\mathcal{I}_0)|) + H(|\text{ADOM}(\mathcal{I}_0)|)$. $P_i(\cdot)$ is a polynomial, and therefore the claim is proven.

In the above claim, i is bounded by the maximum rank r, which is a constant. Hence, there exists a fixed polynomial $P(\cdot)$ such that the number of distinct values that can exist in the active domains of the run τ is bounded by $P(|\text{ADOM}(\mathcal{I}_0)|)$. Technically, given $\Upsilon_{\mathcal{S}^+} = \langle \mathcal{C}, \mathcal{R}, \Sigma, s_0, db, \Longrightarrow\rangle$, we have:

$$|\bigcup_{s \text{ state of } \tau} db(s)| < P(|\text{ADOM}(\mathcal{I}_0)|)$$

which attests that τ is (data) bounded, and consequently that $\mathcal{S}$ is run-bounded. □

C. NONDETERMINISTIC SERVICES

C.2 State-bounded Systems

PROOF OF THEOREM 5.1. We reuse the proof of Theorem 4.1. Recall that the reduction in this proof constructs for every Turing Machine *TM* a DCDS *with deterministic services* $\mathcal{S}$ that simluates the computation of *TM*. It also constructs a propositional safety property Φ such that $\Upsilon_{\mathcal{S}} \models \Phi$ if and only if *TM* halts.

What we need here is a reduction to a DCDS with *nondeterministic services*. However, we recall from the proof of Theorem 4.1 that the only service in the process layer, service $newCell$, is guaranteed to be called only with distinct arguments across distinct transitions, and so its behavior is unaffected by the choice of deterministic versus nondeterministic semantics. Therefore, the reduction applies unchanged to DCDS with nondeterministic services. □

PROOF OF THEOREM 5.2. We prove a stronger result, namely for *linear-time* $\mu\mathcal{L}_A$ sentences. Such sentences can be written using LTL syntax.

We reduce from the problem of satisfiability of LTL with freeze quantifier over infinite data words, known to be highly undecidable (Σ^1_1-hard) [20].

Infinite data words [20]. Let Σ be a finite alphabet of labels and D an infinite set of data values. An infinite *data word* $w = \{w_i\}$ is an infinite sequence over $\Sigma \times D$, i.e., each w_i is of the form (a_i, d_i) with $a_i \in \Sigma$ and $d_i \in D$.

LTL with freeze quantifier (LTL$^{\downarrow}$). This logic operates over infinite data words, seen as runs. It extends propositional LTL with a finite number of *registers*, which can record the data value at the current step of the run (position in the data word), and recall it at subsequent steps. The operation of recording the data value at the current position into register i is denoted with $\downarrow_i$. $\uparrow_i$ denotes the boolean comparison of the data value at the current position with the value stored in register i.

As an example, consider the LTL$^{\downarrow}$ sentence

$$\varphi_{ex} = \downarrow_1 X(G(a \implies \neg \uparrow_1))$$

over alphabet $\{a, b\}$, which states that the data value assigned to each label a at positions greater than one is different from the data value at the first position of the data word. Notice that the data value at the first position is recorded in register 1 by operation $\downarrow_1$, and it is compared to subsequent data values by $\uparrow_1$.

The DCDS construction. Given a finite alphabet $\Sigma = \{\sigma_i\}_{i\in\{1,\ldots,n\}}$, we build a DCDS $\mathcal{S} = \langle \mathcal{D}_\Sigma, \mathcal{P}_\Sigma\rangle$ with nondeterministic services, such that each run of $\Upsilon_{\mathcal{S}}$ represents an infinite data word over Σ. In particular, each state in the run holds the label and data value for a single position in the data word. Moreover, given an LTL$^{\downarrow}$ sentence φ over Σ, we construct a $\mu\mathcal{L}_A$ formula Φ, such $\Upsilon_{\mathcal{S}} \models \Phi$ if and only if φ is *unsatisfiable*.

The idea is to model the registers with existentially quantified variables, which $\mu\mathcal{L}_A$ allows us to introduce at any given point in the run and use subsequently, even if in between their binding does not persist in the run.

More precisely, we define the data layer $\mathcal{D}_\Sigma$ of $\mathcal{S}$ as $\mathcal{D}_\Sigma = \langle \mathcal{C}, \mathcal{R}, \emptyset, \mathcal{I}_0\rangle$, where $\mathcal{C} = \Sigma \cup \{0\}$, $\mathcal{R} = \{\text{LABEL}/1, \text{DATUM}/1\}$, and $\mathcal{I}_0 = \emptyset$. Intuitively, LABEL stores the label and DATUM the data value. We then define the process layer $\mathcal{P}_\Sigma$ of $\mathcal{S}$ as $\mathcal{P}_\Sigma = \langle \mathcal{F}, \mathcal{A}_\Sigma, \varrho_\Sigma\rangle$, where:

- $\mathcal{F} = \{f/0\}$.
- For each $1 \in \{1, \ldots, n\}$, ϱ_Σ contains an action α_i with no parameters and no guard ($\mathsf{true} \mapsto \alpha_i$).
- Each $\alpha_i \in \mathcal{A}_\Sigma$ contains a single effect e_i, which creates the position of a data word corresponding to label $\sigma_i \in \Sigma$:

$$e_i : \quad \mathsf{true} \quad \rightsquigarrow \text{LABEL}(\sigma_i) \wedge \text{DATUM}(f())$$

The service call $f()$ is used to get an arbitrary data value from the domain during the action execution. It is nondeterministic, and will therefore return possibly distinct values across the run.

Since actions are always executable, at each step of a run all of them qualify, and one is nondeterministically chosen. In this way, the collection of all runs corresponds to all possible infinite data words. Observe that $\mathcal{S}$ is state-bounded, as each state contains just one label and one data value.

The property. We now define the property. For simplicity of presentation, we show it using an LTL-based syntax (branching is irrelevant here), though it is clearly expressible in $\mu\mathcal{L}_A$.

We obtain φ' from φ by:

1. replacing each freeze quantifier $\downarrow_n$ with $\exists x_n.\textsc{datum}(x_n)$, and
2. replacing each occurrence of $\uparrow_n$ with $\textsc{datum}(x_n)$, and
3. replacing each proposition $\sigma \in \Sigma$ with $\textsc{label}(\sigma)$.

Now let $\Phi := \neg\varphi'$.

We illustrate the rewrite on property φ_{ex} above, obtaining

$$\varphi'_{ex} := \exists x_1 \textsc{datum}(x_1) \wedge \mathbf{X}\,\mathbf{G}(\textsc{label}(a) \implies \neg\textsc{datum}(x_1)).$$

It is easy to see that φ is unsatisfiable over infinite data words using alphabet Σ if and only if $\Upsilon_\mathcal{S} \models \Phi$.

As a result, $\mu\mathcal{L}_A$ verification by state-bounded DCDSs with nondeterministic services is undecidable. □

PROOF OF THEOREM 5.3. See Section C.3. □

C.3 Abstract Transition System

We formalize the discussion from Section 5.3. Since DCDSs with nondeterministic services are modeled by means of transition systems whose states are constituted by database instances, with a slight abuse of notation we will directly use the state to refer to its database instance.

Equality commitments. Consider a set D comprised of constants and of Skolem terms built by applying a Skolem function to constant arguments. An *equality commitment* $\mathcal{H}$ on D is a partition of D, i.e. a set of disjoint subsets of D, called *cells*, such that the union of the cells in $\mathcal{H}$ is D. Moreover, each cell contains at most one constant (but arbitrarily many Skolem terms). For any $e \in D$, $[e]_\mathcal{H}$ denotes the cell e belongs to. The intention of the partition is to model equality and non-equality commitments on the members of D as follows: for every $e_1, e_2 \in D$, $e_1 = e_2$ if and only if $[e_1]_\mathcal{H} = [e_2]_\mathcal{H}$.

Service call evaluations that respect equality commitments. It is convenient to view the concrete transition system $\Upsilon_\mathcal{S}$ in the following equivalent formulation, which emphasizes equality commitments on the service calls: successor states are built by picking an equality commitment $\mathcal{H}$, and then picking a service call evaluation that respects $\mathcal{H}$. More specifically,

- for each state $\mathcal{I}$,
- for each action α,
- for each parameter choice σ, and
- for each equality commitment $\mathcal{H}$ involving the service calls in $\textsc{calls}(\textsc{do}(\mathcal{I}, \alpha, \sigma))$ and the values in $\textsc{adom}(\mathcal{I}) \cup \textsc{adom}(\mathcal{I}_0)$,

$\Upsilon_\mathcal{S}$ contains possibly infinitely many successor states $\mathcal{I}_{next}$, each obtained from $\textsc{do}(\mathcal{I}, \alpha, \sigma)$ by picking a service call evaluation that respects $\mathcal{H}$. We say that evaluation θ *respects* $\mathcal{H}$ if for every two terms $t_1, t_2 \in \textsc{calls}(\textsc{do}(\mathcal{I}, \alpha, \sigma)) \cup \textsc{adom}(\mathcal{I}) \cup \textsc{adom}(\mathcal{I}_0)$, we have $[t_1]_\mathcal{H} = [t_2]_\mathcal{H}$ if and only if $t_1\theta = t_2\theta$.

Given $\mathcal{I}, \alpha, \sigma$ and $\mathcal{H}$, we denote the set of all legal evaluations with

$$\textsc{evals}^\mathcal{H}(\mathcal{I}, \alpha, \sigma) := \{\theta \mid \theta \in \textsc{evals}_\mathcal{C}(\mathcal{I}, \alpha, \sigma), \theta \text{ respects } \mathcal{H}, \textsc{do}(\mathcal{I}, \alpha, \sigma)\theta \models \mathcal{E}\}.$$

Notice that we consider legal only those evaluations that respect the equality commitment $\mathcal{H}$ and that, conforming to the semantics of the concrete transition system, generate successors which satisfy the constraints $\mathcal{E}$. Finally, notice that $\mathcal{H}$ determines an isomorphism type, as all successors of $\mathcal{I}$ generated by the evaluations in $\textsc{evals}^\mathcal{H}(\mathcal{I}, \alpha, \sigma)$ are isomorphic to each other.

Prunings. We observe that for each state $\mathcal{I}$ of the concrete transition system $\Upsilon_\mathcal{S}$, the number of possible choices of α, σ and $\mathcal{H}$ are finite. The sole reason for infinite branching in $\Upsilon_\mathcal{S}$ are the infinitely many distinct evaluations that respect $\mathcal{H}$, whenever $\mathcal{H}$ states that at least one service call result is distinct from $\textsc{adom}(\mathcal{I}) \cup \textsc{adom}(\mathcal{I}_0)$: in that case, the service call can be substituted with any value in $\mathcal{C} \setminus (\textsc{adom}(\mathcal{I}) \cup \textsc{adom}(\mathcal{I}_0))$.

In contrast, we obtain a finitely-branching transition system if instead of keeping the successors generated by *all* evaluations in $\textsc{evals}^\mathcal{H}(\mathcal{I}, \alpha, \sigma, \mathcal{H})$, we keep the successors generated by a *finite subset* of these evaluations (if $\textsc{evals}^\mathcal{H}(\mathcal{I}, \alpha, \sigma)$ is non-empty, we pick a non-empty subset, to ensure that if $\mathcal{H}$ is represented among the successors of $\mathcal{I}$ in $\Upsilon_\mathcal{S}$, it is also represented among the successors of $\mathcal{I}$ in $\Theta_\mathcal{S}$). We call any transition system obtained in this way a *pruning* of $\Upsilon_\mathcal{S}$, and we denote with $\textsc{prunings}(\Upsilon_\mathcal{S})$ the set of all such prunings. By construction, every pruning of $\Upsilon_\mathcal{S}$ is finitely branching.

Formally, let $\mathcal{S}$ be a DCDS and $\Upsilon_\mathcal{S}$ its concrete transition system, with states Σ_C and initial state $\mathcal{I}_0$. A *pruning of* $\Upsilon_\mathcal{S}$ is the restriction of $\Upsilon_\mathcal{S}$ to a subset of states $\Sigma_P \subseteq \Sigma_C$, where Σ_P satisfies the following properties:

(i) $\mathcal{I}_0 \in \Sigma_P$, and

(ii) for each $\mathcal{I} \in \Sigma_C$ and each equality commitment $\mathcal{H}$, if $\mathcal{H}$ is represented by some successor of $\mathcal{I}$ in $\Upsilon_\mathcal{S}$, it is also represented by a successor of I in $\Theta_\mathcal{S}$. We say that $\mathcal{H}$ is *represented* by successor $\mathcal{I}'$ of $\mathcal{I}$ if there exist α, σ and $\theta \in \textsc{evals}^\mathcal{H}(\mathcal{I}, \alpha, \sigma)$ such that $\langle \mathcal{I}, \alpha\sigma\theta, \mathcal{I}' \rangle \in \text{N-EXEC}_\mathcal{S}$.

(iii) for each $\mathcal{I} \in \Sigma_C$, the number of successors of $\mathcal{I}$ that are also in Σ_P is finite.

Clearly, a concrete transition system $\Upsilon_\mathcal{S}$ admits (potentially infinitely) many prunings, but we show next that they all are persistence-preserving bisimilar to $\Upsilon_\mathcal{S}$ (and therefore to each other, due to transitivity of the $\sim$ relation):

LEMMA C.1. *For every concrete transition system $\Upsilon_\mathcal{S}$ and pruning $\Theta_\mathcal{S} \in \textsc{prunings}(\Upsilon_\mathcal{S})$, we have that $\Theta_\mathcal{S} \sim \Upsilon_\mathcal{S}$.*

The result follows from the fact that state isomorphism implies persistence-preserving bisimilarity. In the following, we denote with $s \mapsto_h s'$ the fact that h is an isomorphism from state s to s'.

LEMMA C.2. *Consider a concrete transition system $\Upsilon_\mathcal{S}$ with initial state s_0 and one of its prunings $\Theta_\mathcal{S}$. Let s_C be a state of $\Upsilon_\mathcal{S}$ and s_P a state of $\Theta_\mathcal{S}$. If there exists function h such that h fixes $\textsc{adom}(s_0)$ and $s_P \mapsto_h s_C$, then $s_P \sim_h s_C$.*

PROOF OF LEMMA C.2. Let $\Upsilon_\mathcal{S} = \langle \mathcal{C}, \mathcal{R}, \Sigma_C, s_0, db, \Longrightarrow_C \rangle$ and $\Theta_\mathcal{S} = \langle \mathcal{C}, \mathcal{R}, \Sigma_P, s_0, db, \Longrightarrow_P \rangle$. The proof follows from the following claim:

> **Claim 1.** Given $s_C \in \Sigma_C$ and $s_P \in \Sigma_P$, if $s_P \mapsto_h s_C$ and h is the identity on $\textsc{adom}(\mathcal{I}_0)$, then for each s'_C such that $s_C \Longrightarrow_C s'_C$ there exist s'_P and h' such that
> (i) $s_P \Longrightarrow_P s'_P$;
> (ii) h' is an extension of $h\,|_{\textsc{adom}(s_P) \cap \textsc{adom}(s'_P)}$;
> (iii) h' is the identity on $\textsc{adom}(\mathcal{I}_0)$;

(iv) $s'_P \mapsto_{h'} s'_C$.

Indeed, this claim allows us to exhibit the bisimilarity relation

$$R = \{(x, i, y) \mid x \in \Sigma_P, y \in \Sigma_C, x \mapsto_i y\}.$$

R is a bisimilarity relation because it satisfies the forth condition in the definition of persistence-preserving bisimilarity by Claim 1. It trivially satisfies the back condition because P is constructed by picking a subset of the states of $\Upsilon_{\mathcal{S}}$. Since by construction $(s_P, h, s_C) \in R$, we have $s_P \sim_h s_C$.

To prove Claim 1, we observe that the successor s'_C of s_C is generated by a particular choice of the action α (with condition-action rule $Q \mapsto \alpha$), the parameter instantiation σ_C (such that $s_C \models Q\sigma_C$), the equality commitment $\mathcal{H}_C$ on $\textsc{calls}(\textsc{do}(s_C, \alpha, \sigma_C)) \cup \textsc{adom}(s_C) \cup \textsc{adom}(\mathcal{I}_0)$, and the service call evaluation $\theta_C \in \textsc{evals}^{\mathcal{H}_C}(s_C, \alpha, \sigma_C)$: $s'_C = \textsc{do}(s_C, \alpha, \sigma_C)\theta_C$. We show how to construct σ_P, $\mathcal{H}_P$ and $\theta_P \in \textsc{evals}^{\mathcal{H}_P}(s_P, \alpha, \sigma_P)$ such that $s_P \models Q\sigma_P$ and $s'_P = \textsc{do}(s_P, \alpha, \sigma_P)\theta_P$ satisfies the claim.

We let $\sigma_P = h^{-1}(\sigma_C)$, observing that since Q is a first-order query, it is preserved under isomorphism, so $s_C \models Q\sigma_C$ implies $s_P \models Q\sigma_P$. Thus, σ_P is a legal parameter instantiation.

To construct $\mathcal{H}_P, \theta_P$, we first show that $\bar{s}_C = \textsc{do}(s_C, \alpha, \sigma_C)$ and $\bar{s}_P = \textsc{do}(s_P, \alpha, \sigma_P)$ are isomorphic, as witnessed by the function $\bar{h} : \textsc{adom}(\bar{s}_P) \to \textsc{adom}(\bar{s}_C)$ defined as follows:

$$\begin{aligned} \bar{h} \;:=\; & \{c \mapsto h(c) \mid c \in \textsc{adom}(s_P) \cup \textsc{adom}(\mathcal{I}_0)\} \\ \cup\; & \{f(m_P, \ldots, m_n) \mapsto f(h(m_P), \ldots, h(m_n)) \mid \\ & \quad f(m_P, \ldots, m_n) \in \textsc{calls}(\bar{s}_P)\}. \end{aligned}$$

From the definition of $\bar{h}$ and the fact that the service calls are generated by queries preserved under isomorphism, it follows immediately that $\bar{s}_P \mapsto_{\bar{h}} \bar{s}_C$. It is easy to see that $\bar{h}$ is also an isomorphism between $\textsc{calls}(\textsc{do}(s_C, \alpha, \sigma_C)) \cup \textsc{adom}(s_C) \cup \textsc{adom}(\mathcal{I}_0)$ and $\textsc{calls}(\textsc{do}(s_P, \alpha, \sigma_P)) \cup \textsc{adom}(s_P) \cup \textsc{adom}(\mathcal{I}_0)$, (i.e. $\bar{h}$ preserves the structure of Skolem terms), and therefore between the sets of corresponding equality commitments.

We therefore pick $\mathcal{H}_P = \bar{h}^{-1}(\mathcal{H}_C)$. By construction of P, each equality type is represented among a state's successors in P, i.e. there exists θ_P that respects $\mathcal{H}_P$, and there exists $s'_P \in \Sigma_P$, such that $s'_P = \bar{s}_P\theta_P$.

The existence of legal choices for $\alpha, \sigma_P, \mathcal{H}_P$ and θ_P proves item (i) of Claim 1, namely that $s_P \Longrightarrow_P s'_P$.

To prove the remaining items, we exhibit h' defined as follows:

$$h'(t) := \bar{h}(\bar{t})\theta_C,$$

for some choice of $\bar{t}$ such that $\bar{t}\theta_P = t$.

To see why h' is well-defined, observe that, by construction of the successor states in $\Upsilon_{\mathcal{S}}$, for each $t \in \textsc{adom}(s'_P)$ there must exist $\bar{t} \in \textsc{adom}(\bar{s}_P)$ such that θ_P evaluates $\bar{t}$ to t ($\bar{t}\theta_P = t$). Moreover, observe that if there are distinct $\bar{t}, \bar{u} \in \textsc{adom}(\bar{s}_P)$ such that $t = \bar{t}\theta_P = \bar{u}\theta_P$, it does not matter which one we pick in the definition of h', since $\bar{h}(\bar{t})\theta_C = \bar{h}(\bar{u})\theta_C$. This is because θ_P respects $\mathcal{H}_P$, and therefore $[\bar{t}]_{\mathcal{H}_P} = [\bar{u}]_{\mathcal{H}_P}$. Since $\mathcal{H}_P \mapsto_{\bar{h}} \mathcal{H}_C$, it follows that $[\bar{h}(\bar{t})]_{\mathcal{H}_C} = [\bar{h}(\bar{u})]_{\mathcal{H}_C}$, and since θ_C respects $\mathcal{H}_C$, we have that $\bar{h}(\bar{t})\theta_C = \bar{h}(\bar{u})\theta_C$.

Items (ii), (iii) and (iv) of Claim 1 follow by similar reasoning from the fact that service call evaluations respect the equality commitments, which are isomorphic. □

PROOF OF LEMMA C.1. This is a corollary of Lemma C.2.

Indeed, by definition, $\Theta_{\mathcal{S}} \sim \Upsilon_{\mathcal{S}}$ holds if and only if the initial state s_0^P of $\Theta_{\mathcal{S}}$ is bisimilar to the initial state s_0^C of $\Upsilon_{\mathcal{S}}$, i.e. there exists isomorphism h such that $s_0^P \sim_h s_0^C$.

By definition, a concrete transition system shares the initial state with all its prunings, so $s_0^P = s_0^C$. The identity mapping id witnesses isomorphism: $s_0^P \mapsto_{id} s_0^C$. By Lemma C.2, we have $s_0^p \sim_{id} s_0^C$. □

Eventually Recycling Prunings. While all prunings of a concrete transition system are finitely-branching, they are not guaranteed to be finite. The reason is that they don't necessarily rule out infinitely long simple runs τ, along which the service calls return in each state $\mathcal{I}$ "fresh" values, i.e. values distinct from all values appearing in $\mathcal{I}$ and its predecessors on τ. Towards addressing this problem, we focus on prunings in which the evaluations are not chosen arbitrarily.

Given a finite run τ ending in state $\mathcal{I}$ of $\Upsilon_{\mathcal{S}}$, an action α, a parameter choice σ and an equality commitment $\mathcal{H}$ on $\textsc{calls}(\textsc{do}(\mathcal{I}, \alpha, \sigma))$, we say that evaluation $\theta \in \textsc{evals}^{\mathcal{H}}(\mathcal{I}, \alpha, \sigma)$ *recycles from* τ if each value in the range of θ occurs in τ. We say that pruning $\Theta_{\mathcal{S}}$ is *eventually recycling* if every (finite or infinite) path τ in $\Theta_{\mathcal{S}}$ contains only finitely many states generated by non-recycling evaluations. Formally, if $\tau = s_0 s_1 \cdots$ and the service call evaluation used in $s_i \Longrightarrow_C s_{i+1}$ is denoted as θ_i, then there are only finitely many indexes j such that θ_j does not recycle from $\tau[j]$.

LEMMA C.3. *Let $\Upsilon_{\mathcal{S}}$ be a concrete transition system.*

(i) All eventually recycling prunings of $\Upsilon_{\mathcal{S}}$ are finite.

(ii) If $\Upsilon_{\mathcal{S}}$ is state-bounded, then it has at least one eventually recycling pruning.

PROOF OF LEMMA C.3. *(i):* All eventually recycling prunings are finite.

Let $\Theta_{\mathcal{S}}$ be an eventually recycling pruning of concrete transition system $\Upsilon_{\mathcal{S}}$. By virtue of being a pruning, $\Theta_{\mathcal{S}}$ is finitely branching. We show next that every simple path in $\Theta_{\mathcal{S}}$ has finite length, which together with finite branching implies finiteness by König's Lemma.

Towards a contradiction, assume that there exists infinite simple run τ in $\Theta_{\mathcal{S}}$. Since $\Theta_{\mathcal{S}}$ is eventually recycling, there is a finite prefix of τ such that all values occurring in τ occur also in this prefix. Therefore, τ contains only finitely many distinct values, and hence only finitely many distinct states (databases of given schema over these values). If τ has infinite length, then a pigeonhole argument contradicts the assumption that τ is simple.

(ii): If $\Upsilon_{\mathcal{S}}$ is state-bounded, then it has an eventually recycling pruning.

Let $\Theta_{\mathcal{S}}$ be a pruning obtained from $\Upsilon_{\mathcal{S}}$ by picking the finite subset of evaluations $SE \subseteq \textsc{evals}^{\mathcal{H}}(s, \alpha, \sigma)$ as follows: if there is a run τ in $\Theta_{\mathcal{S}}$ from s_0 to s such that $\textsc{evals}^{\mathcal{H}}(s, \alpha, \sigma)$ includes at least one evaluation that recycles from τ, then SE contains exclusively recycling evaluations (i.e. for each evaluation $\theta \in SE$, there is a run τ from s_0 to s in $\Theta_{\mathcal{S}}$ such that θ recycles from τ). Otherwise, SE is an arbitrary finite subset of $\textsc{evals}^{\mathcal{H}}(s, \alpha, \sigma)$).

We prove that pruning $\Theta_{\mathcal{S}}$ is eventually recycling. By definition, if $\Upsilon_{\mathcal{S}}$ is state-bounded then $|\textsc{adom}(s)| \leq b$ for each state s, where b is the size bound on the state. Assume towards a contradiction that $\Theta_{\mathcal{S}}$ contains a run $\tau = s_0 s_1 s_2 \cdots$ that includes infinitely many states generated by evaluations that do not recycle from τ. It follows that there must exist a finite $k \geq 0$ such that $|\bigcup_{i=0}^{k} \textsc{adom}(s_i)| > 3b$ and such that $\textsc{adom}(s_{k+1})$ contains at least one fresh value, i.e. $\textsc{adom}(s_{k+1}) - \bigcup_{i=0}^{k} \textsc{adom}(s_i) \neq \emptyset$. Let $\theta_{k+1} \in \textsc{evals}^{\mathcal{H}}(s_k, \alpha, \sigma)$ be the service call evaluation that generates s_{k+1}. Clearly θ_{k+1} does not recycle from τ, since it

contains at least one fresh value in its range. However observe that, since the k-length prefix of τ contains at least $3b$ distinct values, this prefix contains at least b values that are distinct from the values in $\text{ADOM}(\mathcal{I}_0) \cup \text{ADOM}(s_k)$ (since by state-boundedness, $|\text{ADOM}(\mathcal{I}_0) \cup \text{ADOM}(s_k)| \leq 2b$). Call the set of these values $\mathcal{V}$.

Also by state-boundedness, θ_{k+1} introduces at most b fresh values. Any one of the values in $\mathcal{V}$ can be used instead of the fresh values introduced by θ_{k+1}, to obtain another evaluation θ^r_{k+1} that respects $\mathcal{H}$. Hence θ_r witnesses an evaluation in $\text{EVALS}^{\mathcal{H}}(s_k, \alpha, \sigma)$ that *does* recycle from τ. But this contradicts the definition of $\Theta_{\mathcal{S}}$, which mandates that θ_{k+1} be dropped in favor of θ^r_{k+1}. □

This result implies that if $\Upsilon_{\mathcal{S}}$ is state-bounded, then there exists a finite-state abstract transition system $\Theta_{\mathcal{S}}$ that is persistence-preserving bisimilar to $\Upsilon_{\mathcal{S}}$. Indeed, any eventually recycling pruning of $\Upsilon_{\mathcal{S}}$ can play the role of $\Theta_{\mathcal{S}}$ (it is finite by Lemma C.3(i), it is bisimilar to $\Upsilon_{\mathcal{S}}$ by Lemma C.1, and one is guaranteed to exist by Lemma C.3(ii)).

Construction of Eventually Recycling Pruning. The existence result in Lemma C.3 is non-constructive and therefore does not yet yield decidability of verification even if the concrete transition system $\Upsilon_{\mathcal{S}}$ is state-bounded. We next present Algorithm RCYCL, which is guaranteed to construct an eventually recycling pruning when its input DCDS is state-bounded, but which may diverge otherwise.

Algorithm RCYCL
Input: $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$, a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$.

$\Sigma := \{\mathcal{I}_0\}$, $\Longrightarrow := \emptyset$, $\mathit{UsedValues} := \text{ADOM}(\mathcal{I}_0)$, $\mathit{Visited} := \emptyset$
repeat
  pick state $\mathcal{I} \in \Sigma$, action α and legal parameters σ such that $(\mathcal{I}, \alpha, \sigma) \notin \mathit{Visited}$
  $\mathit{RecyclableValues} := \mathit{UsedValues} - (\text{ADOM}(\mathcal{I}_0) \cup \text{ADOM}(\mathcal{I}))$
  pick set $\mathcal{V}$ of n service call results such that:
    $|\mathcal{V}| = n = |\text{CALLS}(\text{DO}(\mathcal{I}, \alpha, \sigma))|$ and
    if $|\mathit{RecyclableValues}| \geq n$
      then $\mathcal{V} \subseteq \mathit{RecyclableValues}$ % recycled values
      else $\mathcal{V} \subset \mathcal{C} - \mathit{UsedValues}$ % fresh values

  $F := \text{ADOM}(\mathcal{I}_0) \cup \text{ADOM}(\mathcal{I}) \cup \mathcal{V}$
  for each $\theta \in \text{EVALS}_F(\mathcal{I}, \alpha, \sigma)$ such that $\mathcal{I}_{next} \models \mathcal{E}$ where $\mathcal{I}_{next} := \text{DO}(\mathcal{I}, \alpha, \sigma)\theta$ **do**
    $\Sigma := \Sigma \cup \{\mathcal{I}_{next}\}$
    $\Longrightarrow := \Longrightarrow \cup \{(\mathcal{I}, \mathcal{I}_{next})\}$
    $\mathit{UsedValues} := \mathit{UsedValues} \cup \text{ADOM}(\mathcal{I}_{next})$
    $\mathit{Visited} := \mathit{Visited} \cup \{(\mathcal{I}, \alpha\sigma)\}$
  end
until Σ and $\Longrightarrow$ no longer change.
return $\langle \mathcal{C}, \mathcal{R}, \Sigma, \mathcal{I}_0, \Longrightarrow \rangle$

Observe that algorithm RCYCL performs several nondeterministic choices in each iteration. The particular choices (and their order) do not matter, by Theorem 5.4.

PROOF OF THEOREM 5.4. *(Sketch)*
First, we show that algorithm RCYCL builds a pruning. Items (i) and (iii) in the definition of pruning are trivially satisfied in every run of RCYCL. Item (ii) follows from the following claim:

> **Claim:** for any choice of $\mathcal{V}$ such that $|\mathcal{V}| \geq |\text{CALLS}(\text{DO}(\mathcal{I}, \alpha, \sigma))|$, the set of equality commitments represented by the successors of $\mathcal{I}$ generated by the evaluations in $\text{EVALS}_F(I, \alpha, \sigma)$ coincides with the set of commitments represented by the successors of $\mathcal{I}$ in $\Upsilon_{\mathcal{S}}$.

Next, we show that if $\mathcal{S}$ is state-bounded, every run of RCYCL terminates. Indeed, state-boundedness guarantees that in each iteration, only at most b service call values are needed, where b is the state size bound. But after running "sufficiently" long, RCYCL variable $\mathit{UsedValues}$ accumulates at least $3b$ distinct values. At each subsequent step of the algorithm, there will therefore exist at least b values distinct from the active domains of $\mathcal{I}_0$ and $\mathcal{I}$, so the pick of $\mathcal{V}$ will always recycle values (observe that RCYCL only picks evaluations from set $\Theta_{\mathcal{S}}$). $\mathit{UsedValues}$ will no longer change, and therefore Σ and $\Longrightarrow$ must eventually saturate (a key reason for this is the bookkeeping of variable $\mathit{Visited}$, which avoids repeating the nondeterministic pick for any combination of state, action and parameter instantiation $(\mathcal{I}, \alpha, \sigma)$).

Finally, since RCYCL terminates, then it outputs a finite-state pruning, which is trivially eventually recycling. □

Theorem 5.4 and Theorem 3.2 directly imply Theorem 5.3.

C.4 GR-Acyclic DCDSs

PROOF OF THEOREM 5.5. For the proof, we reduce from the undecidable problem of checking if the run of a deterministic Turing Machine is confined to a bounded-length segment of the tape (we say that the TM is tape-bounded). This in turn is undecidable by reduction from the halting problem: Given deterministic TM T, build TM T' such that T' is tape-bounded if and only if T halts. T' simulates T but also records on the tape the historical configurations of T. At each step, T' checks if the most recent configuration of T was seen in the history. If so, T' stops simulating T and enters a loop in which it keeps extending the right end of its tape. It is easy to see that T' is tape-bounded if and only if T halts.

We reuse without change the reduction exhibited in the proof of Theorem 4.1. Recall that the reduction constructs for every Turing Machine *TM* a DCDS with deterministic services $\mathcal{S}$ that simluates the computation of *TM*. We recall from the proof that the only service in the process layer, service $newCell$, is guaranteed to be called only with distinct arguments across distinct transitions, and so its behavior is unaffected by the choice of deterministic versus nondeterministic semantics. We also note that the state of the DCDS has size linear in the length of the tape segment visited by *TM*, so tape-boundedness reduces to state-boundedness. □

PROOF OF THEOREM 5.6 (SKETCH). We prove the result by counting the maximum number of different values in a state of the transition system.

Since this task is undecidable (by Theorem 5.5), we necessarily have to approximate this value. The approximation is performed by analysing a different, much more abstract transition system we call *dataflow* transition system (to distinguish from the abstract system that is bisimilar to the concrete system).

The dataflow system is a DCDS obtained as follows from the dataflow graph and $\mathcal{I}_0$: For each node of the dataflow graph, there is a unary relation in the dataflow system, and for each normal (special) edge in the dataflow graph, there is a normal (special) transition in the dataflow system between the corresponding relations. The schema of the dataflow system is a set of relation names with arity one, in correspondence to the nodes of the dataflow graph. A state of the dataflow system is an instantiation of its schema using values from the domain $\mathcal{C}$.

For each term t appearing in a relation in the initial state of the concrete system, there is a term t in the corresponding relation of

the initial state of the dataflow system. Being in one state of the dataflow system, the next state is constructed as follows:

- for each normal transition from a relation A to a relation B, for each term t in the relation A of the current state, there is a term t in the relation B of the next state.
- for each special edge from a relation A to a relation B, for each term t in the node A of the current state, there is a fresh term t' in node B of the next state.

It is easy to see the following claim:

> **Claim 1.** For any run τ of length $m \geq 0$ in the concrete system, there is a run τ^d of length m in the dataflow system, such that the size of the active domain of state $\tau(i)$ is at most the size of the active domain of state $\tau^d(i)$.

As a result, any state bound for the dataflow system also bounds the state of the concrete system. We compute such a bound next.

Consider the dataflow graph of $\mathcal{A}$. GR-acyclicity forces cycles with special edges to not be connected to any other cycles in the dataflow graph. More specifically, each connected component of the dataflow graph must have one of the following types:

A: A simple cycle C (possibly with special edges), possibly connected with several directed acyclic graphs (DAG)s, such that the component contains no additional cycle beyond C.
B: Several cycles $C_1, \ldots, C_m$ containing only normal edges, each C_i possibly connected to several DAGs, such that the component contains no other cycle beyond the C_i's, and there is no path with special edges connecting two cycles C_i, C_j.
C: A DAG, possibly containing normal and special edges.

Denote with

d: the longest path of the dataflow graph after deleting the cycles,
b: the maximum number of special edges going out of a node of the dataflow graph plus one, and
n: the number of nodes of the dataflow graph.

It is easy to see that in each transition of the dataflow system, for each term in the current state, there can be at most $n \cdot b$ distinct terms in the next state.

First, consider the components of type A. Call the DAGs D connected to the unique cycle C via edges from D to C, *input DAGs*. Call *output DAGs* the DAGs connected via edges from C to D. It is easy to see that after d transition steps, in any run of the dataflow system there is no term in any relation of an input DAG (all have been forgotten), and at most

$$m := |\textsc{adom}(\mathcal{I}_0)| + n \cdot b \cdot |\textsc{adom}(\mathcal{I}_0)| + \cdots + n^d \cdot b^d \cdot |\textsc{adom}(\mathcal{I}_0)|$$

distinct terms may co-exist within the relations of the cycle. Moreover, after d steps, the total number of distinct terms in the cycle will no longer increase in any run suffix starting from step $d+1$. Consider now how the m terms can be copied into the output DAGs. It is again easy to see that there can be at most $n^d \cdot b^d \cdot m$ distinct terms in any relation of an outgoing DAG. As a result, in any step at most $n^{d+1} \cdot b^d \cdot m$ distinct terms may co-exist within a type A component.

Second, consider the components of type B. A similar argument yields at most $n^{d+1} \cdot b^d \cdot m$ different terms that may co-exist within a type B component.

Third, it is easy to see that there can be at most $n^d \cdot b^d \cdot |\textsc{adom}(\mathcal{I}_0)|$ different terms within a type C component.

All in all, at most $|\textsc{adom}(\mathcal{I}_0)| \cdot n^{2d+1} \cdot b^{2d}$ distinct terms may co-exist in a state of the concrete transition system. □

D. DISCUSSION

PROOF OF THEOREM 6.1 (SKETCH). The technical problem here is to force the results of nondeterministic service calls to conform to historic evaluations.

Let D be a deterministic DCDS. We rewrite D to obtain a new DCDS N whose semantics under nondeterministic services coincides with that of D under deterministic services. For each term $f(a_1, \ldots, a_n)$ appearing in some effect of $D\ e := q^+ \wedge Q^- \rightsquigarrow E$, we rewrite D as follows. We extend the schema with a new $n+1$-ary relation R_f. Intuitively, $R_f(a_1, \ldots, a_n, r)$ states that the call $f(a_1, \ldots, a_n)$ evaluates to r. We extend the effect to record this fact, replacing e with : $e' := q^+ \wedge Q^- \rightsquigarrow E \wedge R_f(a_1, \ldots, a_n, f(a_1, \ldots, a_n))$. To ensure that R_f records all past calls of f, we add to each action an effect that simply copies R_f. We also add the functional dependency $a_1, \ldots, a_n \rightarrow r$ on R_f. Notice that any attempt to record a service call with a result distinct from a past invocation violates the functional dependency and the transition does not occur. It is easy to see that, if we project the states of Υ_N on the schema of D, we obtain Υ_D. □

PROOF OF THEOREM 6.2 (SKETCH). The challenge here lies in forcing a deterministic service f_d to return possibly distinct results for same-argument calls of the nondeterministic service f_n it corresponds to.

The trick is to call f_d with one additional argument, which plays the role of a timestamp, where each state in the run has its own unique timestamp. This way same-argument calls of f_n at distinct steps in the run correspond to distinct-argument calls of f_d, which therefore simulates the desired nondeterministic behavior.

An additional trick is used to get the run to generate a sequence of unique values to act as timestamps. We use a deterministic service $new(x)$ to generate a new timestamp, we record the successor relation over timestamps in binary relation $succ$, and the most recent timestamp in unary relation now. We add to each action

- the effect
$$now(x) \rightsquigarrow now(new(x)) \wedge succ(x, new(x))$$
which extends the successor relation by one timestamp and sets the new timestamp as most recent; and
- the effect
$$succ(x, y) \rightsquigarrow succ(x, y)$$
which accumulates the historical *succ* entries.

We are not quite done, as we still need to ensure that *succ* induces a linear order on the collection of timestamps generated during the run. For this purpose we employ the same trick as the proof of Theorem 4.1. We describe it below for the sake of proof self-containment.

Observe that, by definition of the effect extending *succ*, at each step, the generated timestamp has at most one successor.

However, if the call $new(x)$ returns a previously seen timestamp, then there can be some timestamp with several predecessors in *succ*. We rule out this case by declaring the second component of *succ* to be a key. It follows that *succ* must be either (i) a linear path over timestamps (possibly starting from a source node that has a self-loop), or (ii) must contain a simple cycle involving more than one timestamp. The simple cycle is created when *new* returns the minimal element of the *succ* relation.

We wish to force case (i). To rule out case (ii), we proceed as follows: we initialize *succ* to contain a source node 0 that can never be a timestamp because it cannot be returned by *new* without violating the key constraint on *succ*. To this end, we initialize *succ* in

$\mathcal{I}_0$ to $succ^{\mathcal{I}_0} = \{(0,0),(0,1)\}$ and $now^{\mathcal{I}_0} = \{1\}$. Notice that $succ^{\mathcal{I}_0}$ has type (i). An easy induction shows that every run prefix must also construct a *succ* relation of type (i), since any attempt to extend *succ* with an edge back to one of its existing nodes violates the key constraint. It also follows easily that during the run, *now* takes values from the linear path starting at 1, and never includes 0. □

E. EXAMPLE DCDS: TRAVEL REIMBURSEMENT SYSTEM

We model the process of reimbursing travel expenses in a university, and the corresponding audit system, in two different subsystems. In particular, the first subsystem, called the *request system* manges the submission of reimbursement requests by an employee, and preliminary inspection and approval of the request by an employee in the accounting department (we shall call her the *monitor*). A log of accepted requests will be submitted to the second subsystem, the *audit system*, in which requests can be accumulated, and they can be checked for accuracy by calling external web services (for instance to obtain the exchange rate from foreign currency to USD on a past date, or to check that the employee actually was on the declared flight).

Request system. To keep the example simple we model a travel reimbursement request as being associated to the name of the requester, and infor- mation related to the corresponding flight and hotel costs. After a request is submitted, a monitor will check the request and will decide to accept or reject the request. If a request is rejected, the employee needs to modify the information regarding hotel and flight, while employee name will not be changed while updating. After the update by the employee, the monitor will again check the request, and the reject-check loop continues until the monitor accepts the request. After a request is accepted a log of the request will be sent to the audit system, and the request system will be ready to process the next travel request.

We model the request system by a DCDS $\mathcal{S}_R = \langle \mathcal{D}, \mathcal{P} \rangle$, in which $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ such that $\mathcal{C}$ is a countably infinite set of *constants*, $\mathcal{I}_0 = \{\mathsf{Status}(\text{`}\textit{readyForRequest}\text{'}), \mathsf{true}\}$, and $\mathcal{R}$ is a database schema as follows:

- $\mathsf{Status} = \langle \mathsf{status} \rangle$, a unary relation that keeps the state of the request subsystem, and can take three different values: *'readyForRequest'*, *'readyToVerify'*, and *'readyToUpdate'*,
- $\mathsf{Travel} = \langle \mathsf{eName} \rangle$, holding the name of the employee;
- $\mathsf{Hotel} = \langle \mathsf{hName}, \mathsf{date}, \mathsf{price}, \mathsf{currency}, \mathsf{priceInUSD} \rangle$, holding the hotel cost information of the employee's travel, which might have been paid in some other currency than USD,
- $\mathsf{Flight} = \langle \mathsf{date}, \mathsf{fNum}, \mathsf{price}, \mathsf{currency}, \mathsf{priceInUSD} \rangle$, holding the flight cost information.

The process layer is defined as $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$ where $\mathcal{F}$ is a set of the following nondeterministic service calls, each modeling an input of an external value by the employee. Specifically,

- INENAME() : models the input of the employee name (filled in by the employee),
- INHNAME() : hotel name,
- INHDATE() : arrival date,
- INHPRICE() : sum paid to the hotel (possibly in foreign currency),
- INHCURRENCY() : currency exchange rate at that date,
- INHPINUSD() : amount paid to the hotel in USD,
- INFDATE() : flight date,
- INFNUM() : flight number,
- INFPRICE() : ticket price, possibly in foreign currency,
- INFCURRENCY() : currency exchange rate at date of ticket payment,
- INFPUSD() : ticket price in USD.

There is one additional service.

- MAKEDECISION() : a nondeterministic service modeling the decision of the human monitor. It returns *'requestConfirmed'* if the request is accepted, and returns *'readyToUpdate'* if the request needs to be updated by the employee.

The set $\mathcal{A}$ of actions includes (among others):

InitiateRequest :

$$\begin{aligned}
&\mathsf{true} \rightsquigarrow \mathsf{Status}(\text{`}\textit{readyToVerify}\text{'})\\
&\mathsf{true} \rightsquigarrow \mathsf{Travel}(\text{INENAME}())\\
&\mathsf{true} \rightsquigarrow \mathsf{Hotel}(\text{INHNAME}(), \text{INHDATE}(), \text{INHPRICE}(), \text{INHCURRENCY}(), \text{INHPINUSD}())\\
&\mathsf{true} \rightsquigarrow \mathsf{Flight}(\text{INFDATE}(), \text{INFNUM}(), \text{INFPRICE}(), \text{INFCURRENCY}(), \text{INFPUSD}())
\end{aligned}$$

VerifyRequest:

$$\begin{aligned}
\mathsf{true} &\rightsquigarrow \mathsf{Status}(\text{MAKEDECISION}())\\
\mathsf{Travel}(n) &\rightsquigarrow \mathsf{Travel}(n)\\
\mathsf{Hotel}(x_1,\ldots,x_5) &\rightsquigarrow \mathsf{Hotel}(x_1,\ldots,x_5)\\
\mathsf{Flight}(x_1,\ldots,x_5) &\rightsquigarrow \mathsf{Flight}(x_1,\ldots,x_5)
\end{aligned}$$

UpdateRequest:

$$\begin{aligned}
\mathsf{true} &\rightsquigarrow \mathsf{Status}(\text{`}\textit{readyToVerify}\text{'})\\
\mathsf{Travel}(n) &\rightsquigarrow \mathsf{Travel}(n)\\
\mathsf{true} &\rightsquigarrow \mathsf{Hotel}(\text{INHNAME}(), \text{INHDATE}(), \text{INHPRICE}(), \text{INHCURRENCY}(), \text{INHPINUSD}())\\
\mathsf{true} &\rightsquigarrow \mathsf{Flight}(\text{INFDATE}(), \text{INFNUM}(), \text{INFPRICE}(), \text{INFCURRENCY}(), \text{INFPUSD}())
\end{aligned}$$

AcceptRequest:

$$\mathsf{Status}(\text{`}\textit{requestConfirmed}\text{'}) \rightsquigarrow \mathsf{Status}(\text{`}\textit{readyForRequest}\text{'})$$

When a request is initiated (modeled by the action *InitiateRequest*), (i) the system changes state "to waiting for verification", (ii) a travel event is generated and the employee fills in his name, (iii) the employee fills in all hotel information, and (iv) the employee fills in all flight information.

Action *VerifyRequest* models the preliminary check by the monitor. Travel event, hotel and flight information are unchanged,

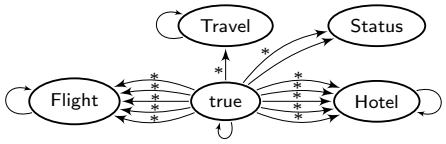


Figure 9: Dataflow graph for request system

but the system status is set by the non-deterministic service call MAKEDECISION(), which models the monitor's decision for current active travel information.

If the monitor rejects, then she sets the next state to *'readyToUpdate'*, which will trigger the action *UpdateRequest*, which in turn collects once again the hotel and flight information from the employee, and moves the status to *'readyToVerify'*.

Finally, action *AcceptRequest* returns the system in the state *'readyForRequest'*, in which it is ready to accept a new request.

Notice the use of the always true predicate true, with the evident meaning. A convenient way to model its meaning in the DCDS framework is to think of it as a nullary relation, initialized to contain the empty tuple, which is copied in perpetuity by each action (true never changes its interpretation). We omit the corresponding copy effects, treating them as built-in.

Notice how the condition-action rules in the set ϱ below guard the actions by the current state of the system:

$$\begin{aligned} \mathsf{Status}\,(\text{'}\textit{readyForRequest}\text{'}) &\mapsto \textit{InitiateRequest} \\ \mathsf{Status}\,(\text{'}\textit{readyToVerify}\text{'}) &\mapsto \textit{VerifyRequest} \\ \mathsf{Status}\,(\text{'}\textit{readyToUpdate}\text{'}) &\mapsto \textit{UpdateRequest} \\ \mathsf{Status}\,(\text{'}\textit{requestConfirmed}\text{'}) &\mapsto \textit{AcceptRequest} \end{aligned}$$

The dataflow graph corresponding to the request system is depicted in Figure 9, where special edges are starred. Notice that there can be multiple special edges between the same two nodes (these are distinguished by unique edge ids, which we omit in the figure to avoid clutter).

In particular, the group of special edges from the true node to the Hotel node corresponds to the action of employee filling in the hotel information, modeled by calls to such services as INHNAME(). Similarly for the special edges from true to Flight.

The special edge between true and Travel is due to the employee filling in his name into the created travel request. The special edge from true to Status reflects the monitor's insertion of her decision (see the call to MAKEDECISION() in the first effect of action *VerifyRequest* in Example E), while the normal edge corresponds to change of the status without calling a service (this happens in other actions). The self-loops on Flight, Hotel, and Travel are due to the remaining (copy) effects of *VerifyRequest*. The self-loop on node true is due to the modeling of this value by a singleton nullary relation containing the empty tuple, which keeps being copied in each action.

An inspection of this dataflow graph reveals that the request system is not GR-acyclic, since it contains several instances of two simple cycles connected by a path that includes a special edge. For instance, the path π comprised of the self-loops around true and Travel, and the special edge beetween them. However, the request system is GR$^+$-acyclic. To illustrate this, notice that the path π is allowed by GR$^+$-acyclicity because the special edge leading into the Travel loop is caused by action *InitiateRequest*, while all the subsequent edges in π are caused by other actions (in this case there is only one subsequent edge in π, namely the self-loop on Travel, caused by actions *VerifyRequest* and *UpdateRequest*).

We illustrate some $\mu\mathcal{L}_P$ properties pertaining to the proper operation of the request system:

A property of interest is that once initiated, a request will eventually be decided by the monitor, and the decision can only be *'readyToUpdate'* or *'requestConfirmed'* (a liveness property). We show the property in the easier-to-read CTL syntactic sugar:

$$\begin{aligned} \mathbf{AG}(\forall n\ \mathsf{Travel}(n) \rightarrow \\ \mathbf{A}(\mathsf{Travel}(n)\mathbf{U}(\mathsf{Status}(\text{'}\textit{readyToUpdate}\text{'}) \vee \\ \mathsf{Status}(\text{'}\textit{requestConfirmed}\text{'})) \end{aligned}$$

The until operator **U** (for this example, it is the strong flavor, in which $\psi\mathbf{U}\phi$ means that ϕ is guaranteed to eventually hold, and until it does ψ must hold in every step). We note that for a property to belong to $\mu\mathcal{L}_P$, it must require the bindings of quantified variables to be continuously live between the step when the quantification was evaluated and the step when the variable is used. This can be done by using LIVE or by using any relation, in our example Travel. The $\mu\mathcal{L}_P$ version of the property is given below:

$$\begin{aligned} &\nu X.(\forall n.\mathsf{Travel}(n) \rightarrow \\ &\quad \mu Y.(\mathsf{Status}(\text{'}\textit{readyToUpdate}\text{'}) \vee \mathsf{Status}(\text{'}\textit{requestConfirmed}\text{'}) \\ &\qquad \vee [-](\mathsf{Travel}(n) \wedge Y))) \wedge [-]X \end{aligned}$$

Another property of interest is that if the flight cost is not specified, then the request is not accepted (a safety property). We use the special constant $\perp$ to denote a null value (this need not be treated specially in the semantics, any value of the domain can be reserved for this purpose):

$$\begin{aligned} &\mathbf{G}\neg(\ \mathsf{Status}(\text{'}\textit{requestConfirmed}\text{'}) \wedge \\ &\quad \exists x_1, \ldots, x_4\ \mathsf{Flight}(x_1, x_2, \perp, x_3, x_4)). \end{aligned}$$

The $\mu\mathcal{L}_P$ version is given below:

$$\begin{aligned} &\nu X.\{\neg(\mathsf{Status}(\text{'}\textit{requestConfirmed}\text{'}) \wedge \\ &\qquad \exists x_1, \ldots, x_4.\mathsf{Flight}(x_1, x_2, \perp, x_3, x_4))\} \wedge [-]X \end{aligned}$$

Audit system. After a request is verified by the monitor in the request system, it will be migrated to the audit system. The migration is performed by a logging subsystem which might perform such operations as: we extend each travel event with a freshly generated travel id, which guarantees uniqueness across the entire history of requests. We store these tuples in a database. We can model this migration using the DCDS formalism, but we omit the specification and focus directly on the audit system.

More specifically, we model the audit system by a DCDS $\mathcal{S}_A = \langle \mathcal{D}_A, \mathcal{P}_A \rangle$, in which $\mathcal{D}_A = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$. $\mathcal{C}$ is a countably infinite set of *constants*. $\mathcal{R}$ is a database schema as follows:

- $\mathsf{Status} = \langle \mathsf{status} \rangle$ is a unary relation keeping the state of request subsystem, which can take two different values: *'checkPrice'*, and *'checkTravel'*, whose role is to sequence the actions of the audit system appropriately.
- $\mathsf{Travel} = \langle \underline{\mathsf{id}}, \mathsf{eName}, \mathsf{passed} \rangle$ extends the homonymous relation of the request system with two fields: id (the travel identifier), and passed, which will be set by the audit system to reflect whether both the hotel and the flight price checks succeed.
- $\mathsf{Hotel} = \langle \mathsf{trId}, \mathsf{hName}, \mathsf{date}, \mathsf{price}, \mathsf{currency}, \mathsf{priceInUSD}, \mathsf{passed} \rangle$, where trId is a foreign key to the travel id and passed is set by the audit system to reflect whether the claimed price and the calculated price match.
- $\mathsf{Flight} = \langle \mathsf{trId}, \mathsf{fNum}, \mathsf{date}, \mathsf{price}, \mathsf{currency}, \mathsf{priceInUSD}, \mathsf{passed} \rangle$, where trId and passed are analogous to the ones in the Hotel relation.

Finally, $\mathcal{I}_0$ is the output of the logging subsystem to which we add the fact $\mathsf{Status}(\text{`}\textit{checkPrice}\text{'})$, to initialize the audit system status.

The process layer is defined as $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$ in which $\mathcal{F}$ contains a deterministic service, where the call $\text{CONVERTANDCHECK}(price, currency, date, priceInUSD)$ perfoms the official exchange rate acquisition and computation described above, returning true if and only if the claimed price and the computed one match.

$\mathcal{A} = \{\textit{CheckPrice}, \textit{CheckTravel}\}$ includes the following actions.

CheckPrice :

$$\mathsf{true} \rightsquigarrow \mathsf{Status}(\text{`}\textit{checkTravel}\text{'})$$

$$\mathsf{Travel}(i, n, v) \rightsquigarrow \mathsf{Travel}(i, n, v)$$

$$\mathsf{Hotel}(x_1, x_2, date, price, currency, priceInUSD, x_7) \rightsquigarrow \mathsf{Hotel}(x_1, x_2, date, price, currency, priceInUSD, \text{CONVERTANDCHECK}(date, price, currency, priceInUSD))$$

$$\mathsf{Flight}(x_1, x_2, date, price, currency, priceInUSD, x_7) \rightsquigarrow \mathsf{Flight}(x_1, x_2, date, price, currency, priceInUSD, \text{CONVERTANDCHECK}(date, price, currency, priceInUSD))$$

Notice that the first effect changes the audit system's state to enter the stage in which the the two checks (for hotel and flight) are combined. The second effect simply copies the request information. The third and fourth each check the claimed price (for hotel, respectively flight), performing the conversion described above.

The second action works on the result of the first (this is ensured by the appropriate status changes).

CheckTravel:

$$\mathsf{true} \rightsquigarrow \mathsf{Status}(\text{`}\textit{checkPrice}\text{'})$$

$$\mathsf{Travel}(x_1, x_2, x_3) \wedge \mathsf{Hotel}(x_1, y_1 \ldots, y_5, p_h) \wedge \mathsf{Flight}(x_1, z_1 \ldots, z_7, p_f) \wedge \neg(p_h \wedge p_f) \rightsquigarrow \mathsf{Travel}(x_1, x_2, \mathsf{false})$$

$$\mathsf{Travel}(x_1, x_2, x_3) \wedge \mathsf{Hotel}(x_1, y_1 \ldots, y_5, \mathsf{true}) \wedge \mathsf{Flight}(x_1, z_1 \ldots, z_7, \mathsf{true}) \rightsquigarrow \mathsf{Travel}(x_1, x_2, \mathsf{true})$$

$$\mathsf{Hotel}(x_1, \ldots, x_7) \rightsquigarrow \mathsf{Hotel}(x_1, \ldots, x_7)$$

$$\mathsf{Flight}(x_1, \ldots, x_7) \rightsquigarrow \mathsf{Flight}(x_1, \ldots, x_7)$$

Notice that the second and third effects set the passed field for the request, computed as the conjunction of the corresponding fields set by the price check on flight and hotel.

The process ϱ is defined as follows:

$$\mathsf{Status}(\text{`}\textit{checkPrice}\text{'}) \mapsto \textit{CheckPrice}$$
$$\mathsf{Status}(\text{`}\textit{checkTravel}\text{'}) \mapsto \textit{CheckTravel}$$

The corresponding dependency graph is as shown in Figure 10. In this picture nodes correspond to the positions of the schema. To avoid clutter, we represent each relation by its first letter, and denote the position number with a subscript. For instance, T_1 stands for the first (id) position of the relation Travel, and S stands for the only position of the relation Status. Moreover, the edges without label represent regular edges in the dependency graph, while the starred edges depict special edges. For instance, the edge $F_5 \xrightarrow{*} F_7$ is introduced due to the fourth effect of action *CheckPrice*. It is starred because it reflects the service call of CONVERTANDCHECK, taking as argument the currency attribute of Flight (at position 5), and storing its result in an Flight tuple at position 7 (the passed attribute).

An inspection of the dependency graph reveals that the audit system is weakly acyclic, since there is no cycle including a special edge.

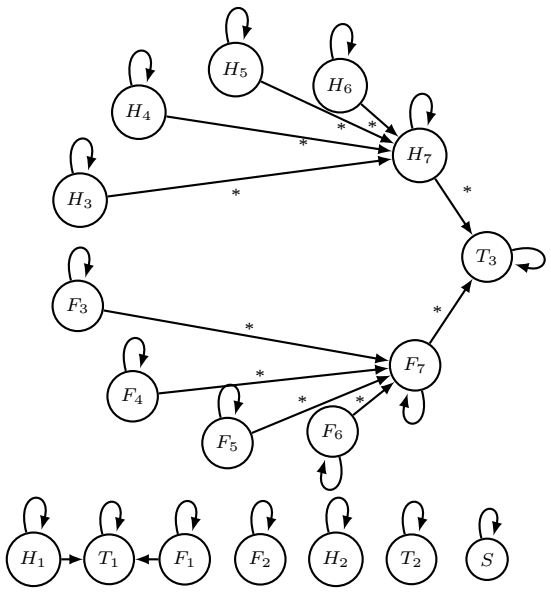


Figure 10: Weakly-acyclic dependency graph of the audit system

We illustrate a desirable property of the audit system: it guarantees that the request cannot pass the audit if one of the flight or hotel checks fail:

$$\mathbf{A}\,\mathbf{G}(\exists i, n, v, x_2, \ldots, x_6.\mathsf{Travel}(i, n, v) \wedge (\mathsf{Hotel}(i, x_2, \ldots, x_6, \mathsf{false}) \vee \mathsf{Flight}(i, x_2, \ldots, x_6, \mathsf{false})) \rightarrow \mathbf{F}\ \mathsf{Travel}(i, n, \mathsf{false}))$$

The $\mu\mathcal{L}_A$ version of the property is given below:

$$\nu X.(\exists i, n, v, x_2, \ldots, x_6.\mathsf{Travel}(i, n, v) \wedge (\mathsf{Hotel}(i, x_2, \ldots, x_6, \mathsf{false}) \vee \mathsf{Flight}(i, x_2, \ldots, x_6, \mathsf{false})) \rightarrow \mu Y.(\mathsf{Travel}(i, n, \mathsf{false}) \vee \langle - \rangle Y)) \wedge [-]X$$

Notice that, since the audit system uses deterministic services, if we wish to verify it in isolation from the other subsystems, we can verify an $\mu\mathcal{L}_A$ property, which is what the above is (we are not enforcing the liveness of the variables i, v, n between the step at which the quantification was evaluated, and the eventual step when the passed attribute of Travel is set to false).

Recall however from Section 6 that we can verify mixed semantics DCDS by reduction to non-deterministic services. If we wished to verify the above property over the collection of subsystems, we

would have to express it as an $\mu\mathcal{L}_P$ property. This is easily done using an until operator **U** (as illustrated above for the request system). Moreover, it is actually compatible with our expectation about the system's operation: while a request is being audited, we expect it to persist in the system.